Chapter 5

High-Performance Arithmetic Units

Computer arithmetic has an intrinsic relationship to technology and the implementation of algorithms in a digital computer, owing to the fact that the value of a particular algorithm is directly related to the actual speed with which this computation is performed. There are other measures of the value of the algorithm: the area of the VLSI chip taken for its implementation, regularity of the implementation, and wireability (the ability to connect the pieces with relatively short and regular interconnections), as well as power consumption, which has recently became a very important feature. A direct and strong relationship exists between the technology in which digital logic is implemented and the way the computation is organized. This relationship is one of the guiding principles in developing computer arithmetic.

Number Representation

Information is represented in a digital computer by a string of bits, that is, zeroes and ones. This relationship is defined by the rule that associates one numerical value designated as X (in the text we will use capital X for the numerical value) with the corresponding bit string designated as x.

$$x = \{x_{n-1}, x_{n-2}, \ldots\ldots, x_0\}$$

where : $x_i \in 0, 1$

In this case the word is n bits long.

When for every value X one and only one corresponding bit string x exists, we define the number system as *nonredundant.* If, however, we can have more than one bit string x that represents the same value X, the number system is said to be *redundant.*

The digit set x_i can be both redundant and nonredundant. If the number of different values x_i can assume is $n \leq r$, then we have a nonredundant digit set; otherwise, if $n > r$, we have a redundant digit set. Use of the redundant digit set has its advantages in efficient implementation of algorithms (multiplication and division in particular).

We also define *explicit value* x_e and *implicit value* X_i of a number represented by a bit-string x. The implicit value is the only value of interest to the user, whereas the explicit value provides the most direct interpretation of the bit-string x. Mapping of the explicit value to the implicit value is obtained by an arithmetic function that defines the number representation used. The arithmetic designer's task is to devise algorithms that result in the correct implicit value of the result for operations on the operand digits representing the explicit values. In other words, the arithmetic algorithm needs to satisfy the *closure* property. The relationship between implicit value and explicit value is illustrated by Table 5.1.

Representation of Signed Integers. The two most common representations of signed integers are Sign and Magnitude (SM) and True and Complement (TC). Although the SM representation might be easier to understand and convert to and from, it has implementation problems. Therefore, we will find that TC representation is more commonly used.

Sign and Magnitude Representation (SM). In SM representation, the signed integer X_i is represented by sign bit x_s and magnitude x_m (x_s, x_m). Usually, 0 represents a positive sign (+), and 1 represents a negative sign (−). The magnitude of the number x_m can be represented in any way chosen for the representation of positive integers. The disadvantage of SM representation is that two representations of zero exist, positive and negative zero: $x_s = 0$, $x_m = 0$, and $x_s = 1$, $x_m = 0$.

True and Complement Representation (TC). TC representation does not use a separate bit to represent the sign. Mapping between the explicit and implicit value is defined as

$$X_i = \begin{cases} x_e & x_e \leq C/2 \\ x_e - C & x_e > C/2 \end{cases}$$

For an illustration of TC mapping, see Table 5.2.

In this representation positive integers are represented in the *true form,* whereas negative are represented in the *complement form.*

TABLE 5.1 The Relationship Between the Implicit Value and Explicit Value for $x = 11011$, $r = 2$

Implied Attributes: Radix Point, Negative Number Representation, Others	Expression for Implicit value X_i as a function of explicit value x_e	Numerical implicit value X_i (in decimal)
Integer, Magnitude	$X_i = x_e$	27
Integer, Two's Complement	$X_i = -2^5 + x_e$	−5
Integer, One's Complement	$X_i = -(2^5 - 1) + x_e$	−4
Fraction, Magnitude	$X_i = 2^{-5} x_e$	27/32
Fraction, Two's Complement	$X_i = 2^{-4}(-2^{-5} + x_e)$	−5/16
Fraction, One's Complement	$X_i = 2^{-4}(-2^{-5} + 1 + x_e)$	−4/16

*A. Avizienis, "Digital Computer Arithmetic: A Unified Algorithmic Specification," Symp. Computers and Automata, Polytechnic Institute of Brooklyn, April 13–15, 1971.

With respect to how the complementation constant C is chosen, we can further distinguish two representations within the TC system:

- If the complementation constant is chosen to be equal to the range of possible values taken by x_e, $C = r^n$ in a conventional number system where $0 \le x_e \le r^n - 1$, then we have defined the *Range Complement* (RC) system.
- If, on the other hand, the complementation constant is chosen to be $C = r^n - 1$, we have defined the *Diminished Radix Complement* (DRC), which is also known as the *Digit Complement* (DC) number system. The RC and DRC number representation systems are shown in Table 5.3.

As can be seen in Table 5.3, the RC system provides for one unique representation of zero because the complementation constant $C = r^n$ falls outside the range. There are two representations of zero in the DRC system, $x_e = 0$ and $r^n - 1$. The RC representation is not symmetrical, and it is not a closed system under the change of sign operation. The range for RC is $[-1/2r^n, 1/2r^n - 1]$. The DRC is symmetrical and has the range of $[-(1/2r^n - 1), 1/2r^n - 1]$.

For the radix $r = 2$ RC and DRC number representations are commonly known as the *Two's Complement* and *One's Complement* number representation systems. These two representations are illustrated Table 5.4 for the range of values $-(4 \le X_i \le 3)$.

TABLE 5.2 True and Complement Mapping

x_e	X_i
0	0
1	1
2	2
—	—
—	—
$C/2 - 1$	$C/2 - 1$
$C/2 + 1$	$-(C/2 - 1)$
—	—
—	—
$C - 2$	−2
$C - 1$	−1
C	0

TABLE 5.3 Mapping of the Explicit Value x_e into RC and DRC Number Representations

x_e	X_i(RC)	X_i(DRC)
0	0	0
1	1	1
2	2	2
—	—	—
—	—	—
$1/2r^n - 1$	$1/2r^n - 1$	$1/2r^n - 1$
$1/2r^n$	$-1/2r^n$	$-(1/2r^n - 1)$
—	—	—
—	—	—
—	—	—
$r^n - 2$	−2	−1
$r^n - 1$	−1	0

Implementation of Elementary Arithmetic Operations

The algorithms for the arithmetic operation are dependent on the number representation system used. Therefore, their implementation should be examined for each number representation system separately, given that the complexity of the algorithm, as well as its hardware implementation, is dependent on it.

Implementation of Fast Addition in VLSI. The first significant speed improvement in the implementation of a parallel adder was the Carry-Lookahead-Adder (CLA) developed by Weinberger and Smith in 1963. Even today the CLA adder is one of the fastest schemes used for the addition of two numbers, given that the delay incurred to add two numbers is logarithmically dependent on the size of the operands (delay = log[N]). The CLA concept is illustrated in Fig. 5.1 *a*, *b*. For each bit position of the adder, a pair of signals (p_i, g_i) is generated in parallel. Local carries can be generated using (p_i, g_i), as seen in the equations. Those signals are designated as: p_i—*carry-propagate* and g_i—*carry-generate,* because they take part in the propagation and generation of carry signal C_{out}. However, each bit position requires an incoming carry signal C_{in} in order to generate the outgoing carry C_o. This makes the addition slow because the carry signal has to ripple from stage to stage as shown in Fig. 5.1*a*.

The adder can be divided into groups, and *carry-generate* and *carry-propagate* signals can be calculated for the entire group

TABLE 5.4 Two's Complement and One's Complement Representations

	Two's Complement	$C = 8$	One's Complement	$C = 7$
X_i	x_e	X_i (2's complement)	x_e	X_i (1's complement)
3	3	011	3	011
2	2	010	2	010
1	1	001	1	001
0	0	000	0	000
−0	0	000	7	111
−1	7	111	6	110
−2	6	110	5	101
−3	5	101	4	100
−4	4	100	3	—

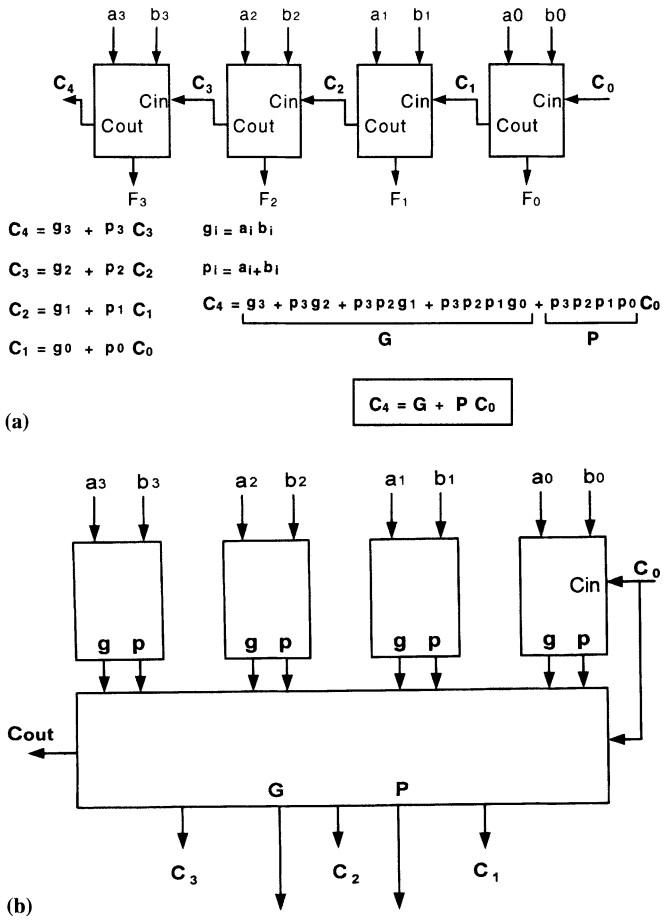


Fig. 5.1. Lookahead Adder structure. **(a)** Generation of carry generate and propagate signals, **(b)** Generation of group signals *G, P* and intermediate carries.

(G, P). This will take an additional time equivalent to and AND-OR delay of the logic. However, now we can calculate each group's carry signals in an additional AND-OR delay. For the generation of the carry signal from the adder, only the incoming carry signal into the group is now required. Therefore, the rippling of the carry is limited to the groups. In the next step we may calculate *generate* and *propagate* signals for the group of groups (G^*, P^*) and continue in that fashion until we have only one group left, generating the C_{out} signal from the adder. This process will terminate in a log number of steps, given that we are generating a tree structure for the generation of carries. The computation of carries within groups is done individually as illustrated in Fig. 5.1*a*, and this process requires only the incoming carry into the group.

The logarithmic dependence on the delay (delay = log[*N*]) is valid only under the assumption that the gate delay is constant without depending on the fan-out and fan-in of the gate. In practice, this is not true, and even when the bipolar technology (which does not exhibit strong dependence on the fan-out) is used to implement the CLA structure, the further expansion of the carry-block is not possible given the practical limitations of the fan-in of the gate.

In CMOS technology, this situation is much different given that the CMOS gate has a strong dependency not only on fan-in but on fan-out as well. This limitation takes away many of the advantages gained by using the CLA scheme. However, by clever optimization of the critical path and appropriate use of dynamic logic, the CLA scheme can still be advantageous, especially for adders of a larger size as demonstrated in the paper by Naini et al.

Conditional-Sum Addition. Another fast scheme for the addition of two numbers, which predates CLA, is the Conditional-Sum Addition method proposed by Sklansky in 1960. The essence of CSA is the realization that we can add two numbers without waiting for the carry signal to be available. Simply put, the numbers are added in two instances: one assuming $C_{in} = 0$ and the other assuming $C_{in} = 1$. The results: Sum^0, Sum^1 and Carry^0, and Carry^1 are presented at the input of a multiplexer. The final values are selected at the time C_{in} arrives at the "select" input of a multiplexer. As in CLA, the input bits are divided into groups, which are added "conditionally." It is apparent that starting from the Least Significant Bit (LSB) position, the hardware complexity starts to grow rapidly. Therefore, in practice, the full-blown implementation of the CSA is seldom seen.

The idea of adding the Most Significant Bit (MSB) portion conditionally and selecting the results once the carry-in signal is computed in the LSB portion is attractive, however. Such a scheme (which is a subset of CSA) is known as Carry-Select Adder. A 26-b Carry-Select Adder consisting of two 13-bit portions is shown in Fig. 5.2.

As can be seen in Fig. 5.2, the block sizes in the particular 13-bit adders used consist of rather unusual numbers: 1–3–5–3–1. Each block is an optimized Carry-Skip Adder (CSA) known as a Variable Block Adder (VBA) as presented in the 1985 paper by Oklobdzija and Barnes. The sizes of the blocks are determined using linear programming techniques, with objective function being the carry signal delay. In this way, not only was the speed of the CSA reduced but also the delay dependency was changed from that of linear in CSA to a square root (delay = Sqrt(*N*)).

Further optimization of the critical path in a VLSI adder can be obtained if the multiple levels of *"skip"* paths are introduced and optimized. Such optimization does not yield a close form solution, however; in some instances, the speed of such an optimized VBA surpasses that of the CLA (with the groups of regular sizes).

The delay of the critical path in the VBA adder is calculated taking fan-in and fan-out (as well as wire delays) into account, making it more realistic for a VLSI implementation. A particular implementation of 16-bit and 32-bit VBA adders is presented in the paper by Oklobdzija and Barnes. The comparison with CLA and adders based on the Recurrence Solving Scheme (HPA) shows that in spite of its hardware simplicity the VBA adder outperforms both: CLA and HPA (for the 32-bit size). This demonstrates that in VLSI the complexity of the algorithm may quickly pass the point of diminishing returns.

The observation that the multilevel VBA scheme outperformed CLA led to an investigation of the CLA and the way the group sizes have traditionally been determined. The paper by Lee and Oklobdzija shows that the CLA (as traditionally known) is a suboptimal structure. This explains why CLA has been outperformed by a multilevel VBA, which is indeed an

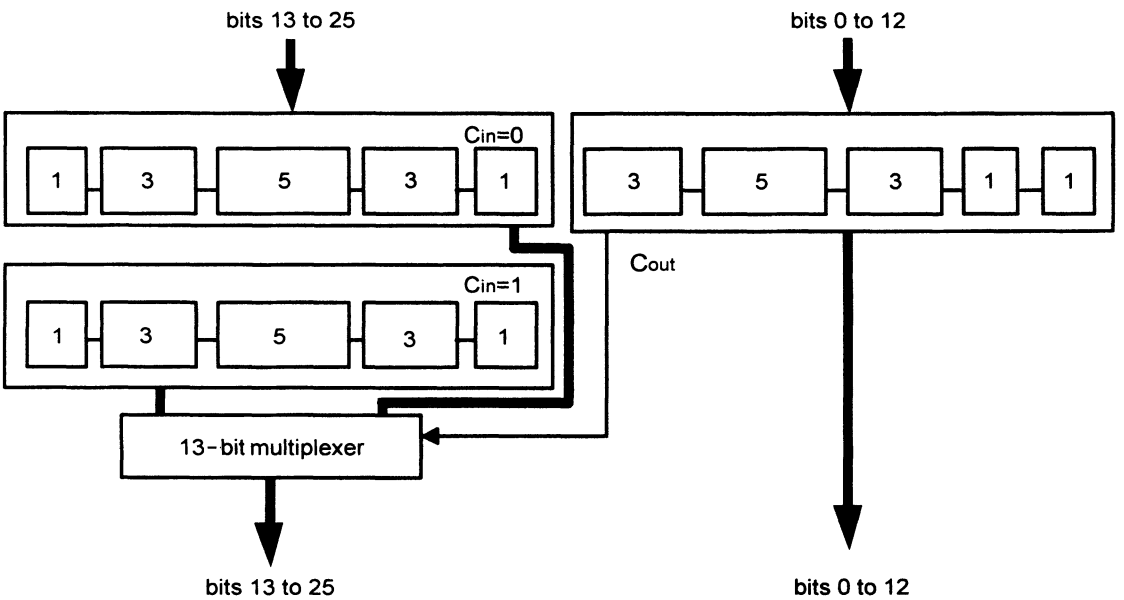


Fig. 5.2. 26-bit Carry-Select Adder.

optimized subset of CLA. The paper by Lee and Oklobdzija also shows how to determine an optimal group size for a speed-optimized CLA. Such optimized CLA outperforms the traditional CLA in terms of speed.

Detection of Leading Zero. In a floating-point arithmetic, normalization is required. In order to "normalize" the number (which consists of shifting the fraction to the left until a nonzero bit is reached), it is necessary to know in advance the number of zeroes before a nonzero bit is reached. Such an operation is known as the *counting of leading-zeroes* and is performed by a circuit known as the Leading Zero Detector (LZD). A LZD circuit consists of a block with N inputs and $\log[N]$ outputs, of which each output is dependent on all of the inputs. When the number of inputs is large (usual case), the design and optimization of such a circuit is not a straightforward problem. Oklobdzija showed how such a design can be treated in a modular and hierarchical way, which greatly simplifies the design of an LZD. In order to determine the structure of the LZD, an algorithm for computing an LZD is devised first. Implementation of such an algorithm from a number of optimized modular blocks not only yields the minimal number of transistors used, but also results in the fastest LZD circuit. Arriving at the design solution in an algorithmic way is important because such an algorithm can be easily integrated into the logic synthesis tools. This helps the logic synthesis tools to produce better and more efficient hardware.

Multiplication. The speed of the multiply operation is of great importance in Digital Signal Processors (DSPs), as well as in general-purpose processors. Therefore, research in building a fast parallel multiplier has been ongoing since C. S. Wallace published the first paper on this subject in 1964. In his historic paper, Wallace outlined a new approach to summing the partial product bits in parallel using a tree of Carry-Save Adders, which later became known as the Wallace Tree shown in Fig. 5.3.

A paper by Dadda suggests a speed improvement of the process of adding partial product bits in parallel. In his 1965 paper, Dadda introduces a notion of a counter structure that will take a number of bits p in the same bit position (of the same *"weight"*) and output a number q which represents the count of ones in the input. Dadda describes a number of ways to compress the partial product bits using such a counter, which later became known as Dadda's counter. Stenzel and Kubitz undertook an extensive study of the use of Dadda's counters in their 1977 study in which they also demonstrate a parallel multiplier built using ROM to implement counters used for partial product summation [Ling, Naffziger].

Amazingly, the quest for an even faster parallel multiplier continued after almost 30 years. But the search for the fastest "counter" did not succeed in yielding a faster partial product summation than that which used the Full-Adder (FA) cell, or "3:2 counter." Therefore, use of the Wallace Tree became almost prevalent in implementing parallel multipliers.

In 1981 Weinberger disclosed a structure that he called the "4-2 carry-save module." This structure contained a combination of FA cells in an intricate interconnection structure, which was yielding faster partial product compression than the use of 3:2 counters. This scheme was further investigated by Santoro, and it became a popular feature in several VLSI implementations. The 4:2 structure compresses five partial product bits into three. However, it is connected in such a way that four of the inputs are coming from the same bit position of the weight j, while one bit is stemming from the neighboring position $j-1$ (known as carry-in) shown in Fig. 5.4. The output of such a 4-2 module consists of one bit in the position j and two bits in the position $j+1$. This structure does not represent a counter (though it became erroneously known as the "4-2 counter") but a "compressor," which would compress four partial-product bits into two (while using one bit laterally connected between adjacent 4-2 compressors).

The efficiency of such a structure is obviously higher in that it reduces the number of partial product bits by one-half. The speed of such a 4-2 compressor has been determined by the

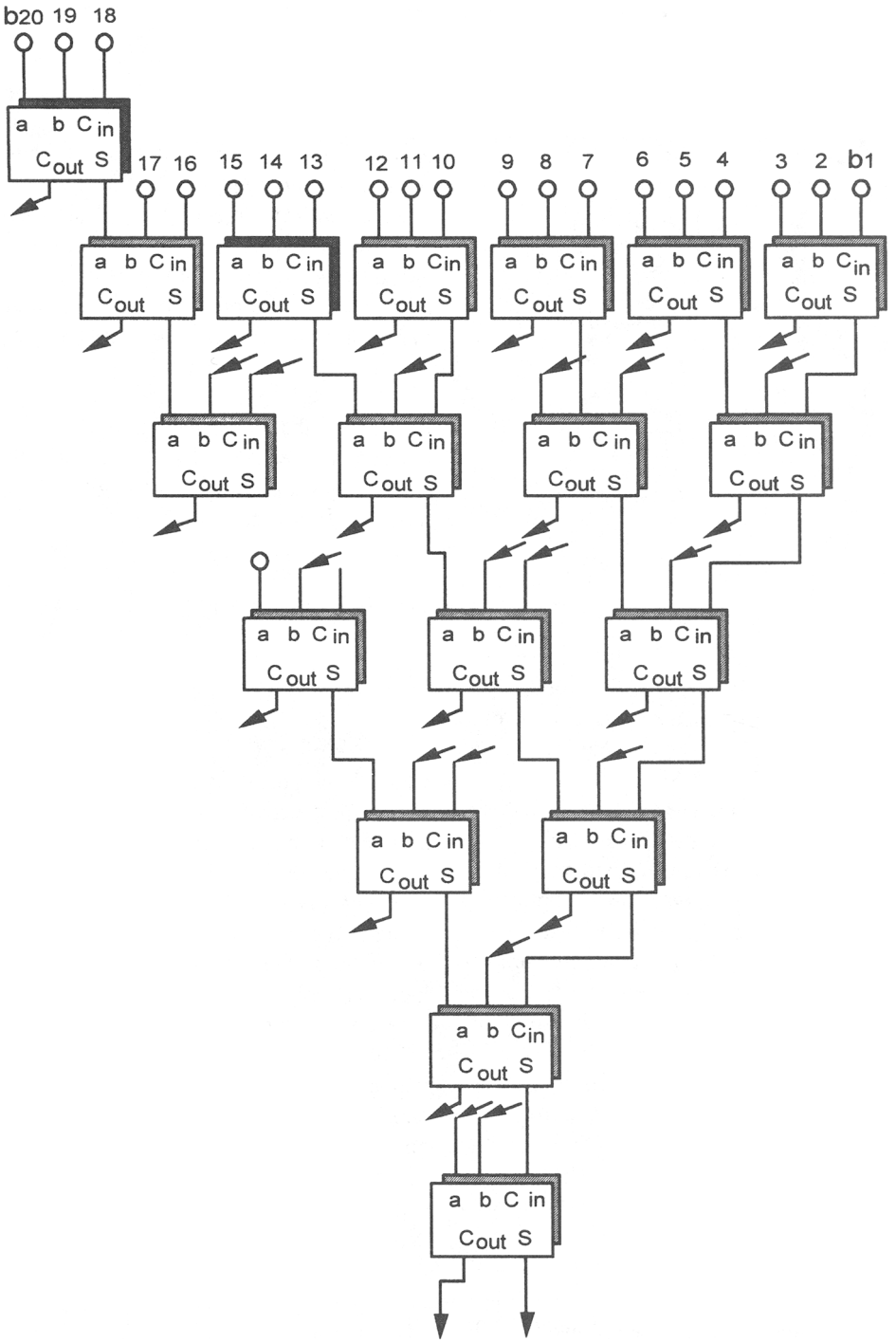


Fig. 5.3. The Wallace Tree.

speed of 3 XOR gates in series (in the redesigned version of 4-2 compressor), making such a scheme more efficient that the one using 3:2 counters in a regular Wallace Tree. The other equally important feature of the 4-2 compressor is that the interconnections between such a cells follow a more regular pattern than in case of the Wallace Tree.

The Booth Recoding Algorithm. Various modifications of the multiplication algorithm exist, but one of the most famous is the Booth Recoding Algorithm described by Booth in 1951. This algorithm allows for the reduction of the number of partial products, thus speeding up the multiplication process. Generally, the Booth algorithm is a case of using the redundant number system with the radix higher than 2.

Booth's algorithm is widely used in the implementation of hardware or software multipliers because its application makes it possible to reduce the number of partial products. It can be used for both sign-magnitude numbers as well as 2's complement number, with no need for a correction term or a correction step.

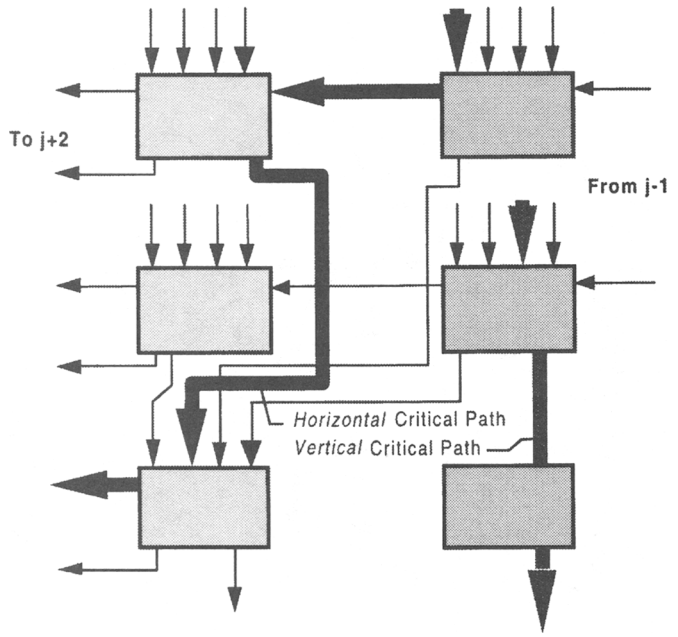


Fig. 5.4. 4-2 compressors.

MacSorley has proposed a modification of the Booth algorithm whereby a triplet of bits is scanned instead of two bits. This technique reduces the number of partial products by one-half regardless of the inputs (see Table 5.5).

Recoding is performed in two steps: *encoding* and *selection.* The purpose of the encoding is to scan the triplet of bits of the multiplier and define the operation to be performed on the multiplicand, as shown in Table 5.5. This method is an application of a sign-digit representation in radix 4. The Booth-MacSorley algorithm, usually called the *Modified Booth algorithm* or simply the *Booth algorithm,* can be generalized to any radix. For example, a 3-bit recoding would require the following set of digits to be multiplied by the multiplicand : $0, \pm 1, \pm 2, \pm 3$. The difficulty lies in the fact that $\pm 3Y$ is computed by summing (or subtracting) Y to $\pm 2Y$, which means that a carry propagation occurs. The delay caused by the carry propagation renders this scheme slower than a conventional one. Consequently, only the 2-bit Booth recoding is used. Booth recoding necessitates the internal use of the 2's complement representation in order to efficiently perform subtraction as well as addition of the partial products. However, the floating-point standard specifies a sign magnitude representation, which is followed by most of the standard floating-point numbers in use today. Booth recoding generates only one-half of the partial products as compared to the multiplier implementation, which does not use Booth recoding. However, this benefit comes at the expense of increased hardware complexity. Indeed, this implementation requires hardware for the *encoding* and *selection* of the partial products $(0, \pm Y, \pm 2Y)$. An optimized *encoding* is shown in Figure 5.5.

TABLE 5.5 Modified Booth Recording

$x_{i+2}x_{i+1}x_i$	Add to Partial product
000	+0Y
001	+1Y
010	+1Y
011	+2Y
100	−2Y
101	−1Y
110	−1Y
111	−0Y

Division. Implementing the division process is more complex because it involves guessing the digits of the quotient. Here, we will consider an algorithm for division of two positive integers designated as *dividend Y*, and *divisor X*, and resulting in a *quotient Q* and an integer *remainder Z* according to the relation given:

$$Y = XQ + Z$$

In this case, the dividend contains $2n$ integers and the divisor has n digits in order to produce a quotient with n digits.

The algorithm for division is given with the following recurrence relationship:

$$z^{(0)} = Y$$

$$z^{(j+1)} = rz^{(j)} - Xr^n Q_{n-1-j} \quad \text{for } j = 0, \ldots, n-1$$

This recurrence relation yields

$$z^{(n)} = r^n(Y - XQ)$$

$$Y = XQ + z^{(n)}r^{-n}$$

which defines the division process with the remainder $Z = z^{(n)}r^{-n}$.

The quotient digit is selected by ensuring that $0 \leq Z < X$ at each step in the division process. This selection is a crucial part of the algorithm, and the best known are *restoring* and *nonrestoring* division algorithms. In the restoring algorithm the value of the *tentative partial remainder* $z^{(j)}$ is restored after the wrong guess is made of the quotient digit q_j. In the nonrestoring, this correction is not done in a separate step, but rather in the step following.

The best known division algorithm is so-called SRT algorithm, which was independently developed by Sweeney, Robertson, and Tocher. Algorithms for a higher radix were further developed by Robertson and his students, most notably Ercegovac.

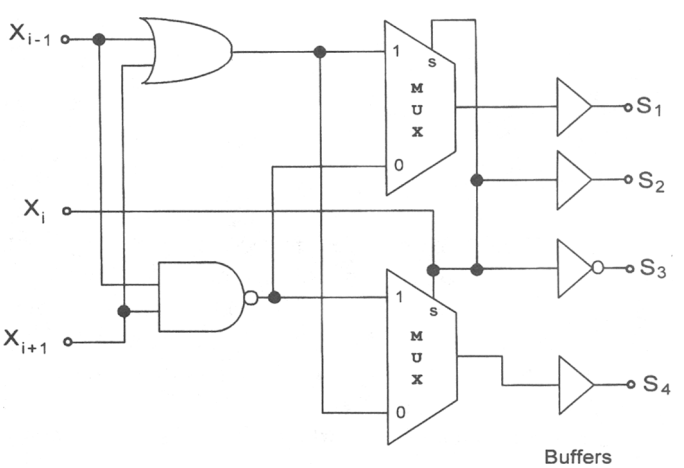


Fig. 5.5. Booth Encoder.

Some Optimal Schemes for ALU Implementation in VLSI Technology

VOJIN G. OKLOBDZIJA AND EARL R. BARNES
IBM T.J. WATSON RESEARCH CENTER
P.O. BOX 218, YORKTOWN HEIGHTS, NY 10598

ABSTRACT

An efficient scheme for carry propagation in an ALU implemented in n-MOS technology is presented. An algorithm that determines the optimum division of the carry chain of a parallel adder for various data path sizes is developed. This yields an implementation of a fast ALU which due to its regular structure occupies a modest amount of silicon. The speed of the implementation described is comparable to the carry look-ahead scheme. Our method is based on the optimization of the carry path implemented in n-MOS technology but the results can be applied to other technologies.

1. INTRODUCTION

An efficient implementation of an ALU in VLSI technology depends on many parameters. We consider an *efficient* implementation to be one which is fast, of a small and regular area, and low power. In many VLSI designs achieving the ultimate speed is not always important goal especially if this is achieved by consuming excessive area and power. Therefore Carry-Lookahead (CLA) scheme is not very attractive for VLSI implementations where area, regularity of structure and power are important. Also the determining factor, in case of an ALU, is wheather it is part of a critical path or not.

In this paper we considered Carry-Skip (CSA) scheme [1],[2],[3],[4] because it met our objective of reasonable performance achieved with a relatively small and regular area. This adder is in essence a Carry Lookahead for which the carry-generate portion which consumes a large amount of logic, has been eliminated. As in a Carry Lookahead adder the bits to be added are divided into groups. A circuit is provided for detecting when a carry signal entering a group will ripple through the group. When this condition is detected, the carry is allowed to skip over the group. Carry-Skip Adder (CSA) does not require excessive amount of logic (area) and the "skip" portion of the logic can be added to the existing carry chain of Ripple-Carry Adder (RCA), therefore not disturbing the inherently regular bit-slice implementation of an RCA (Fig.1.).

The power requirements of a CSA are considerably lower than that of a CLA-ALU. In this paper we show how the carry chain in a CSA can be optimized to yield better speed which in the case of a multi-level optimized CSA-ALU can even outperform the speed of a CLA divided into the groups of constant size.

Lehman and Burla [3] studied a design of a CSA and suggested varying the size of the groups. By varying the sizes of the groups one can influence the maximum delay a carry signal can experience in propagating through the adder. Lehman and Burla [3] posed the problem of determining the optimal group sizes for minimizing the maximum delay [9]. They gave a heuristic method for obtaining economical group sizes. However, they did not solve the problem of determining optimal group sizes. For example, for a 48 bit adder they gave the group sizes 4 5 6 7 8 8 7 6 5 4 yielding the maximum delay of 14 [3]. We show that, under the same assumptions an optimal subdivision results in a delay of 12.

They also discussed the problem of achieving even faster addition by allowing carry signals to skip over blocks of groups. The problem of optimizing the carry chain is now complicated by having to choose both the optimal number of blocks and the optimal sizes of groups within blocks. Some rules for choosing economical block and group sizes are given [3]. However, the problem of determining the optimal sizes remains unsolved.

In this paper we consider the problem of designing a carry-skip adder in FET technology and give some optimal solutions. Actually, our solutions are more general in that we generally assume

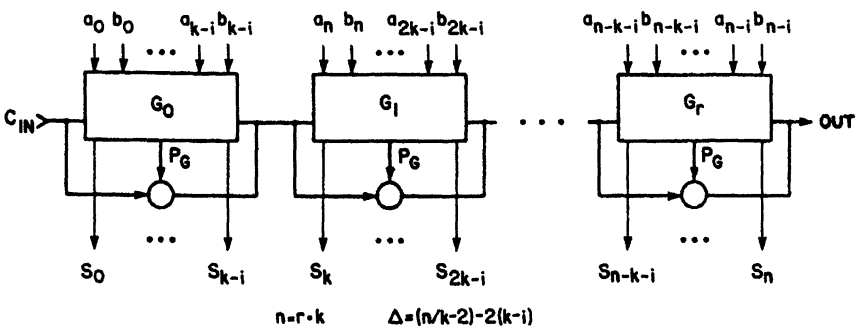


Fig.1. Carry-Skip Adder

Reprinted from *Proceedings of 7th Symposium on Computer Arithmetic,* pp. 2-8, June 1985.

that the time required for a carry signal to skip over a group of bits is longer than the time required for the carry to ripple through a single bit. This assumption is relevant for adders designed in n-MOS technology. Lehman and Burla assumed their adder to be designed using discrete components where these two times are equal. Our analysis will include their problem as a special case.

2. DIVIDING THE ADDER INTO GROUPS

Let n denote the number of bits in a carry skip adder and let m denote the number of groups into which the bits are divided. Let $x_1, \ldots, x_m$ denote the sizes of the groups beginning with the most significant bit. Let T denote the time required for a carry signal to skip over a group of bits. To be precise we should write $T = T(x)$ to indicate that T depends on the size x of the group over which the carry is skipped. However, T changes very slowly with x over the range of group sizes that concern us. So we assume that T is constant.
For a given n, the following three-step procedure gives an optimal way of dividing an n bit adder into groups of bits.

Procedure 2:

2(i) Let m be the smallest positive integer such that

$$n \le m + \frac{1}{2}mT + \frac{1}{4}m^2T + (1 - (-1)^m)\frac{T}{8}. \quad (1)$$

2(ii) Let

$$y_i = \min\{1 + iT,\ 1 + (m + 1 - i)T\}, \quad i = 1, \ldots, m$$

and construct a histogram whose i-th column has height y_i. For example, for $T = 3$ and $n = 48$, we have $m = 7$ and the following histogram.

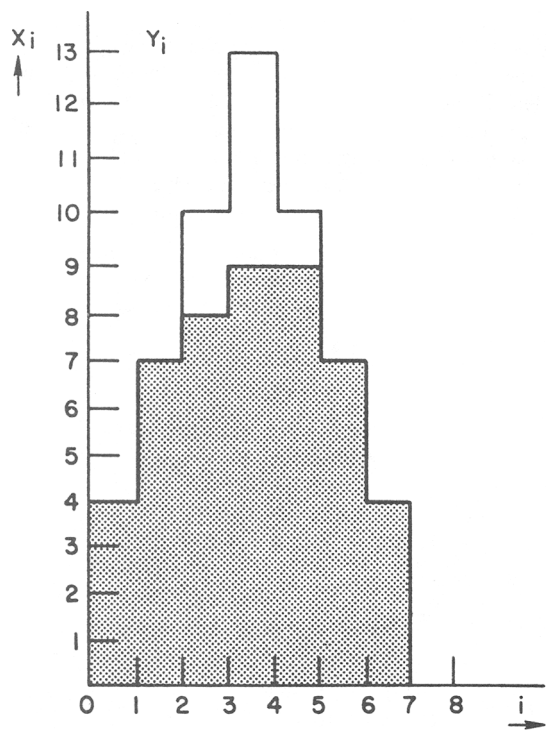


Fig.2. Histogram of segments y_i , x_i

2(iii) It is easily verified that the area of the histogram in (ii) is

$$m + \frac{1}{2}mT + \frac{1}{4}m^2T + (1 - (-1)^m)\frac{T}{8} \ge n$$

so these are at least n unit squares in the histogram. Starting with the first row, shade in n of the squares, row by row. Let x_i denote the number of shaded squares in column i of the histogram, $i = 1, \ldots, m$.

Then $x_1, \ldots, x_m$ is an optimal division of the adder. The maximum delay of a carry signal is mT.

Example 1. For a 48 bit adder we have, from Figure 2. $x_1 = x_7 = 4$, $x_2 = x_6 = 7$, $x_3 = 8$ and $x_4 = x_5 = 9$.

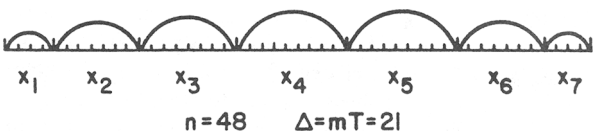


Fig.3. Carry chain of a 48-bit adder: n=48, mT=21

The maximum delay is experienced by a signal generated in the second bit position and terminating in the 47th bit position. The delay is $mT = 21$.

Example 2. Consider a 54 bit adder. From 2(i) we see that again $m = 7$. If we shade 54 squares in Figure 2, we see that

$$x_1 = x_7 = 4,\ x_2 = x_6 = 7,\ x_3 = x_5 = 10 \text{ and } x_4 = 12$$

yields an optimal division of the adder. Again the maximum delay is $mT = 21$.

Example 3. Consider a 64 bit adder. From 2(i) we compute $m = 8$. The corresponding histogram is shown in Figure 4. The optimal group sizes are:

$$x_1 = x_8 = 4,\ x_2 = x_7 = 7,\ x_3 = x_6 = 10,\ x_4 = x_5 = 11.$$

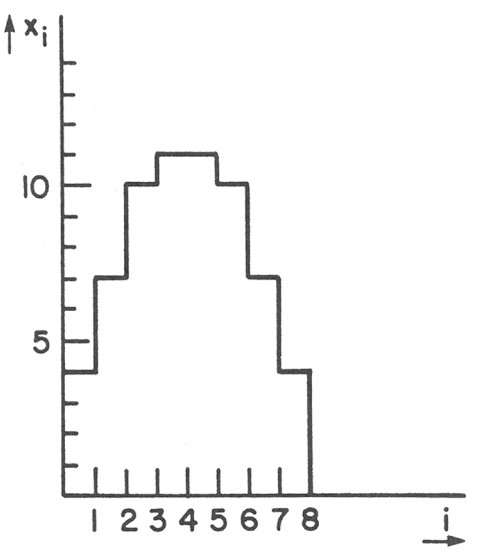


Fig.4. Segment histogram for a 64-bit adder: m=8

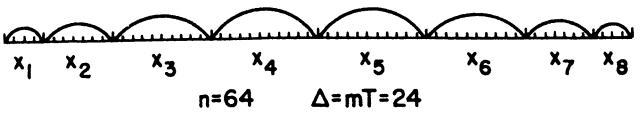


Fig.5. Carry chain of a 64-bit adder: m=8, mT=24

The delay of the longest signal is $mT = 24$.

Proof of Optimality We are going to prove optimality of the division of the carry chain described in 2(i)-(iii). First we need a lemma.

Lemma 1. *When the bits of a carry skip adder are grouped according to the scheme (i)-(iii), the maximum propagation time of a carry signal is mT.*

Proof. The carry generated at the 2nd bit position and terminating at the (n-1) clearly has propagation time mT. We must show that any other signal has propagation time smaller than or equal to mT. Consider a signal originating in the ith group and terminating in the jth, $i < j$. Denote its propagation time by P. Clearly

$$P \le (x_i - 1) + (j - i - 1)T + (x_j - 1).$$

By construction, $x_i \le \min\{1 + iT,\ 1 + (m + 1 - i)T\}$ for each i so

$$\begin{aligned} P \le\ & \min\{1 + iT,\ 1 + (m + 1 - i)T\} \\ & + \min\{1 + jT,\ 1 + (m + 1 - j)T\} + (j - i - 1)T - 2. \end{aligned}$$

There are three cases to consider.

1. First assume

$$\min\{1 + iT,\ 1 + (m + 1 - i)T\} = 1 + iT$$

and

$$\min\{1 + jT,\ 1 + (m + 1 - j)T\} = 1 + jT.$$

In this case $1 + jT \le 1 + (m + 1 - j)T \Rightarrow 2jT - T \le mT$. It follows that

$$\begin{aligned} P &\le 1 + iT + 1 + jT + (j - i - 1)T - 2 \\ &= 2jT - T \le mT. \end{aligned}$$

2. Now assume that

$$\min\{1 + iT,\ 1 + (m + 1 - i)T\} = 1 + iT$$

and

$$\min\{1 + jT,\ 1 + (m + 1 - j)T\} = 1 + (m + 1 - j)T.$$

In this case we have

$$P \le 1 + iT + 1 + (m + 1 - j)T + (j - i - 1)T - 2 = mT.$$

3. Finally, assume that

$$\min\{1 + iT,\ 1 + (m + 1 - i)T\} = 1 + (m + 1 - i)T$$

and

$$\min\{1 + jT,\ 1 + (m + 1 - j)T\} = 1 + (m + 1 - j)T.$$

It follows that

$$\begin{aligned} P &\le 1 + (m + 1 - i)T + 1 + (m + 1 - j)T + (j - i - 1)T - 2 \\ &= 2mT - (2iT - T) \le 2mT - mT = mT. \end{aligned}$$

This completes the proof of the lemma.

Lemma 2. *Let Δ denote the maximum delay of a carry signal in a n bit carry skip adder with group sizes chosen optimally.* Then

$$(m - 1)T \le \Delta \le mT.$$

Proof. Since we have exhibited a division of the carry chain into groups in such a way that the maximum delay of a carry signal is mT we clearly have $\Delta \le mT$.

Let $x_1, x_2, \ldots, x_r$ denote the optimal group sizes corresponding to Δ. For the moment assume that $r = 2k$ is even. By considering carries originating in group i and terminating in group $r - i + 1$, $i = 1, \ldots, k$, we deduce the following system of inequalities.

$$\begin{array}{rcrcl} (x_1 - 1) & + & (r - 2)T & + & (x_r - 1) \le \Delta \\ (x_2 - 1) & + & (r - 4)T & + & (x_{r-1} - 1) \le \Delta \\ \vdots & & & & \vdots \\ (x_k - 1) & + & (r - 2k)T & + & (x_{k+1} - 1) \le \Delta \\ & & rT & & \le \Delta \end{array}$$

If we add these inequalities and use the fact that $\sum_{i=1}^{r} x_i = n$, we obtain the inequality

$$n - 2k + (k + 1)rT - k(k + 1)T \le (k + 1)\Delta$$

which simplifies to

$$\frac{n - 2k}{k + 1} + kT \le \Delta.$$

The left hand side of this inequality assumes its minimum value at

$$k + 1 = \frac{\sqrt{n + 2}}{T}.$$

It follows that

$$\Delta \ge -(T + 2) + \sqrt{4nT + 8T}. \qquad (2)$$

Assume now that $r = 2k + 1$ is odd. We then have the system of inequalities

$$\begin{array}{rcrcl} (x_1 - 1) & + & (r - 2)T & + & (x_r - 1) \le \Delta \\ (x_2 - 1) & + & (r - 4)T & + & (x_{r-1} - 1) \le \Delta \\ \vdots & & & & \vdots \\ (x_k - 1) & + & (r - 2k)T & + & (x_{k+1} - 1) \le \Delta \\ & & (x_{k+1} - 1) & + & kT \le \Delta, \end{array}$$

which implies that

$$\frac{n - 2k - 1}{k + 1} + kT \le \Delta.$$

By minimizing the left hand side of this inequality with respect to k we find that

$$\Delta \ge -(T + 2) + \sqrt{4nT + 4T}. \qquad (3)$$

By comparing this with (2) we see that (3) holds in all cases.

We will now produce an upper bound on mT. Since m is the smallest positive integer satisfying (1) we have

$$n > (m - 1) + \frac{1}{2}(m - 1)T + \frac{1}{4}(m - 1)^2T + (1 - (-1)^{m-1})\frac{T}{8}.$$

Since each size of this inequality is an integer, we can increase the right hand side by 1 to obtain

$$n \geq m - \frac{1}{4}T + \frac{1}{4}m^2T + (1 - (-1)^{m-1})\frac{T}{8}.$$

Solving this inequality for mT gives

$$mT \leq -2 + \sqrt{4nT + 4 + T^2 - (1 - (-1)^{m-1})\frac{T^2}{2}}. \quad (4)$$

Combining this with (2) and (3) gives

$$\Delta \leq (x_i - 1) + sT + (x_j - 1)$$
$$\leq x_i + x_j - 2 + (m-3)T$$

So that either $x_i \geq T + 1$ or $x_j \geq T + 1$. This means that we can subdivide one of the groups i,j without increasing Δ. Continuing in this way, we can always increase the number r of group in an optimal division of a carry chain by 1 without increasing Δ if $r < m$. This means that we can arrive at an optimal division of the carry chain into m groups. We must then have $\Delta \geq mT$ which, together with Lemma 2, implies $\Delta = mT$. This completes the proof of the theorem.

$$mT - \Delta \leq \begin{cases} T + \dfrac{T^2 - 8T + 4 - (1 - (-1)^{m-1})\frac{T^2}{2}}{\sqrt{4nT + 8T} + \sqrt{4nT + 4 + T^2 - (1 - (-1)^{m-1})\frac{T^2}{2}}}, & r\text{-even} \\ T + \dfrac{T^2 - 4T + 4 - (1 - (-1)^{m-1})\frac{T^2}{2}}{\sqrt{4nT + 4T} + \sqrt{4nT + 4 + T^2 - (1 - (-1)^{m-1})\frac{T^2}{2}}}, & r\text{-odd} \end{cases} \quad (5)$$

For n sufficiently large we have $mT - \Delta < T + 1$ and since $mT - \Delta$ is an integer, $mT - \Delta \leq T$. This completes the proof of the lemma.

Theorem 1 *The scheme 2(i)-2(iii) given above for dividing the bits of a carry skip adder into groups is optimal for* $2 \leq T \leq 7$.

Proof. Assume the scheme is not optimal and let Δ be the maximum delay corresponding to an optimal division of the bits into groups. Assume there are r groups in the optimal division. Since a carry in signal to the least significant bit group can skip over each group we have $rT \leq \Delta \leq mT$ so $r \leq m$. If $r = m$ then $\Delta = mT$ and the theorem holds by Lemma 1. If $r = m - 1$ m and r have different parities and it follows from (5) that $mT - \Delta < T$ for $2 \leq T \leq 7$ so that $\Delta > (m-1)T = rT$. This means that a signal which skips over each of the r groups has delay less than the maximum Δ. Similarly, if $r < m - 1$, $\Delta \geq (m-1)T > rT$ so that a signal which skips over each group has delay $< \Delta$. It follows that a signal with delay Δ must start in a group i, ripple to the end of this group, then skip over $s < r$ groups and either terminate, or ripple through the first few bits of a group $j > i$. Let x_i and x_j denote the lengths of the ith and jth groups respectively. Assume that i is chosen as small as possible and j as large as possible. A signal originating in group i, rippling to the end of this group and then skipping over the next s group has delay

$$\Delta \leq (x_i - 1) + sT \leq (x_i - 1) + (r-1)T$$
$$\leq (x_i - 1) + (m-2)T.$$

Since $\Delta \geq (m-1)T$ this implies that $x_i \geq T + 1$. Divide group i into two groups such that the group containing the most significant bits has size T. Since the i-th group is the first group in which a signal having maximum delay can originate, this subdivision does not increase the delay of any carry signal of maximum delay. However, it increases the number of groups by 1.

Suppose now that a carry signal originates in a group i, ripples to its end, skips over $s \leq r - 2$ groups and finally ripples through the first few bits of a group j and terminates. We then have

3. DIVIDING GROUPS INTO BLOCKS

It is clear that the maximum delay of a carry signal in a carry skip adder can be further reduced if signals are allowed to skip over blocks of groups. We define a block to be an additional path allowing carry signal to skip directly over groups. In this section we will describe an efficient scheme for dividing the carry chain into blocks of groups. We assume that the time required for a carry signal to skip over a block of groups is T_b. Actually, the time required for a carry to skip over a block T_b is slightly longer than the time T_g required to skip over a group. But for the sake of simplifying the analysis we will assume these two times to be equal i.e. $T_b = T_g$. However, our technique extends to the case where $T_g \neq T_b$.

Let M denote the number of blocks into which the groups of bits are divided. Let Δ denote the maximum delay a carry signal can have in an adder divided into M blocks. Clearly, $\Delta \geq MT$. We will show how to choose the blocks such that $\Delta = MT$. We will also show how to choose M for an adder of length n.

Our blocks are chosen in such a way that the maximum delay of a signal originating and terminating in block i and $M + 1 - i$ is iT.

Consider a signal originating in the first of these blocks and terminating in the second. Such a signal will skip over $M - 2i$ blocks and will accordingly have delay $\leq (iT) + (M - 2i)T + iT = MT$ as desired. It follows from our work in Section 2 that in order for a signal originating and terminating in block i to have delay less or equal iT we must choose the length of the ith and $(M + 1 - i)th$ blocks to be less or equal the number of unit squares in a histogram with base of width i. Thus the maximum length of the ith and $(M + 1 - i)th$ blocks must be

$$i + \frac{1}{2}iT + \frac{1}{4}i^2T + (1 - (-1)^i)\frac{T}{8}, \quad i \leq \lceil \frac{M}{2} \rceil.$$

(Here we use the symbol $\lceil I \rceil$ to denote the smallest integer $\geq I$.) It follows that the maximum length of an adder divided into M blocks must be

$$2\sum_{i=1}^{\lceil \frac{M-1}{2} \rceil}\left\{i+\frac{1}{2}iT+\frac{1}{4}i^2T+(1-(-1)^i)\frac{T}{8}\right\} \quad (3.1)$$
$$+\frac{(1-(-1)^M)}{2}\left\{\lceil\frac{M}{2}\rceil+\frac{1}{2}\lceil\frac{M}{2}\rceil T+\frac{1}{4}\lceil\frac{M}{2}\rceil^2 T\right.$$
$$\left.+(1-(-1)^{\lceil M/2\rceil})\frac{T}{8}\right\}.$$

Thus for a given adder length n, we choose M to be the smallest positive integer such that the expression (3.1) exceeds or equals n. M is then the number of blocks into which our adder must be divided. The formal statement of our algorithm is as follows.

3(i) Choose M to be the smallest positive integer such that

$$n \le 2\sum_{i=1}^{\lceil \frac{M-1}{2} \rceil}\left\{i+\frac{1}{2}iT+\frac{1}{4}i^2T+(1-(-1)^i)\frac{T}{8}\right\}$$
$$+\frac{(1-(-1)^M)}{2}\left\{\lceil\frac{M}{2}\rceil+\frac{1}{2}\lceil\frac{M}{2}\rceil T+\frac{1}{4}\lceil\frac{M}{2}\rceil^2 T\right.$$
$$\left.+(1-(-1)^{\lceil M/2\rceil})\frac{T}{8}\right\}.$$

3(ii) Form M blocks labeled 1, 2, ... , M, with blocks i and $M+1-i$ each containing

$$i+\frac{1}{2}iT+\frac{1}{4}i^2T+(1-(-1)^i)\frac{T}{8}$$

bits, $i \le \lceil\frac{M}{2}\rceil$.

This construction is analogous to the construction of the histogram in 2(ii). If necessary, delete bits from the largest blocks in this chain until a total of exactly n bits remain in the M blocks.

3(iii) Treat each of the final blocks in 3(ii) as a complete carry chain and divide it into groups optimally using the algorithms 2(i)-2(iii).

Example 3.1. Consider a 32-bit adder. For $i = 1, 2, 3, \ldots$, and $T = 3$ the numbers

$$i+\frac{1}{2}iT+\frac{1}{4}i^2T+(1-(-1)^i)\frac{T}{8}$$

take on the values 4, 8, 15, 22, 32,... respectively. Since

$$32 \le 2\{4+8\}+15$$

we must have $M = 5$ blocks in step 3(ii). These blocks have sizes 4, 8, 15, 8, 4 respectively. If we delete 7 units from the middle block we obtain block sizes 4, 8, 8, 8, 4 which add up to 32. Dividing each block into groups by the procedure 2(i)-2(iii) we obtain the following chain where each group has size 4.
The maximum delay of a carry signal is $MT = 15$.

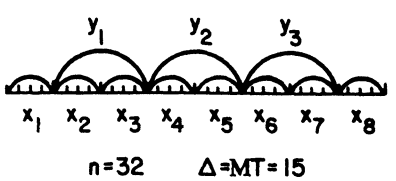


Fig.6. Carry chain of an 32-bit adder: MT=15

Example 3.2. Consider a 48 bit adder. Again we assume $T = 3$. Since

$$48 \le 2\{4+8+15\}$$

we must take $M = 6$ corresponding to block sizes 4, 8, 15, 15, 8, 4. The total number of units is 54. So we reduce the size of the two middle blocks by 3 each. This gives block sizes 4, 8, 12, 12, 8, 4 adding up to 48. If we divide each block into groups by the procedure 2(i)-2(iii), each group has size 4. The maximum delay of a carry signal is given by $MT = 18$.

Example 3.3. Consider a 64 bit adder and assume $T = 3$. Since

$$64 \le 2\{4+8+15\}+22$$

we take $M = 7$ and start with blocks of sizes 4, 8, 15, 22, 15, 8, 4 respectively. The lengths of these blocks total 76. So we reduce the middle block by 12. The new block sizes are 4, 8, 15, 10, 15, 8, 4. The optimal division of these blocks into groups is given in Figure 7. We could have just

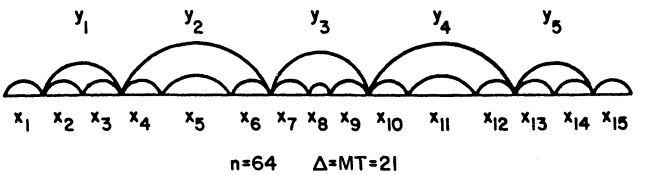


Fig.7. Carry chain of a 64-bit adder: MT = 21

as well reduced the sizes of the three middle blocks obtaining final blocks of sizes 4, 8, 13, 14, 13, 8, 4. The maximum delay of a carry signal would still be $MT = 21$.

4. COMPARISON

In the literature comparison of schemes for ALU implementation is done mainly on the basis of the number of gates, propagation delay per gate and power consumed per gate [5]. However, if a VLSI implementation of a high-speed ALU is considered, these measures are not easily applied. For example, the propagation delay in terms of number of gates is not an adequate measure unless care is taken to implement the function exactly as specified by its logic (gate) representation. This is often not the case since the function is frequently merged into a group of transistors or implemented by using pass-transistors, precharge or other techniques applyed by the circuit designer in order to minimize delay and power.

In general, if the function is implemented in two levels of logic, the delay is not necessarily smaller than the implementation of the same function in three or more logic levels. This is due to the fact that in n-MOS technology the delay is heavily dependent on several factors:

1. *Gate type* : NOR gates are faster than NAND gates.
2. *Fan-in* : for a NAND gate, the delay is directly proportional to the number of inputs, since inside the n-input NAND gate the signal has to propagate through n-transistors. In case of

a NOR gate, delay is not strongly affected by the number of inputs, and therefore the use of NOR gate is preferable.

3. *Fan-out* : the speed of a gate will be different if the fan-out is larger than in the case of small fan-outs.
4. *Wiring* : speed is also dependent on the length of the wires i.e. "wiring capacitance" and the resistance of the long wires.

4.1 Comparison with Carry-Lookahead Scheme

For the purpose of our analysis we consider as a rough measure of delay the number of FET transistors through which the signal needs to propagate in order to reach the destination point. This seems to be satisfactory for measuring delay through a VLSI FET network. In addition we modify the delay equations for the points known to have either a substantial fan-out or considerable capacitive loading due to the long wires carrying signal to the distant points within the VLSI macro block. For example, we assume that the time required for a carry signal to skip over a block of groups is three times the time to ripple through one bit position of the carry-chain. These assumptions were confirmed by simulation performed on a 32-bit ALU which was actually implemented in n-MOS technology.

The Case T = 1.

In order to compare the speed of our carry skip adder with that of the Carry Lookahead Adder (CLA) we take $T = 1$. This modification is made in order to compare our results with the estimates for the CLA adder which are based on the gate delay calculations in which case it is assumed that each gate level introduces a delay $t_G = 1$ regardless of the gate fan-in or fan-out [5]. This assumption works favorably for CLA when calculating its speed. In practice we expect CLA to show worse performance than estimated while the calculation for our CSA schemes should be more accurate.

For our comparison we consider full-CLA with the groups of size $G_s = 4$. CLA adder of size n=32 bits is shown in the Fig.8. with the delays indicated at each signal input and output to or from the block. Delay calculation for CLA of the sizes n = 16, 32, 48, 64 shows delays for the critical path of the carry signal to be Δ = 6, 8, 10, 10 respectively [5].

First we consider our case where the adder is simply divided into groups of bits. The procedure in Section 2 shows that the maximum delay of a carry signal in an n bit adder is m, where m is the smallest integer satisfying

$$n \le m + \frac{1}{2}m + \frac{1}{4}m^2 + (1 - (-1)^m)\frac{1}{8}.$$

The values of a delay m for our method compared with the CLA delays for several values of n are shown in the Table 1.

n	$\Delta(CSA)$	$\Delta(CLA)$
16	6	6
32	9	8
48	12	10
64	14	10

Table 1. Delay comparison with the CLA for the method of Section 2.

Consider now the case where the adder is divided into blocks of groups according to the procedure of Section 3. This algorithm shows that the maximum delay of a carry signal in an n bit carry skip adder is M, the smallest positive integer satisfying

$$n \le 2 \sum_{i=1}^{\lceil \frac{M-1}{2} \rceil} \{i + \frac{1}{2}i + \frac{1}{4}i^2 + (1 - (-1)^i)\frac{1}{8}\} + \frac{(1 - (-1)^M)}{2}\left\{\lceil \frac{M}{2} \rceil + \frac{1}{2}\lceil \frac{M}{2} \rceil + \frac{1}{4}\lceil \frac{M}{2} \rceil^2 + (1 - (-1)^{\lceil M/2 \rceil})\frac{1}{8}\right\}.$$

Recall that $\lceil r \rceil$ is the smallest integer $\ge$ r for any real number r.

Table 2. shows the values of delay M for CSA of Section 3 compared to that of CLA adder corresponding to several values of n.

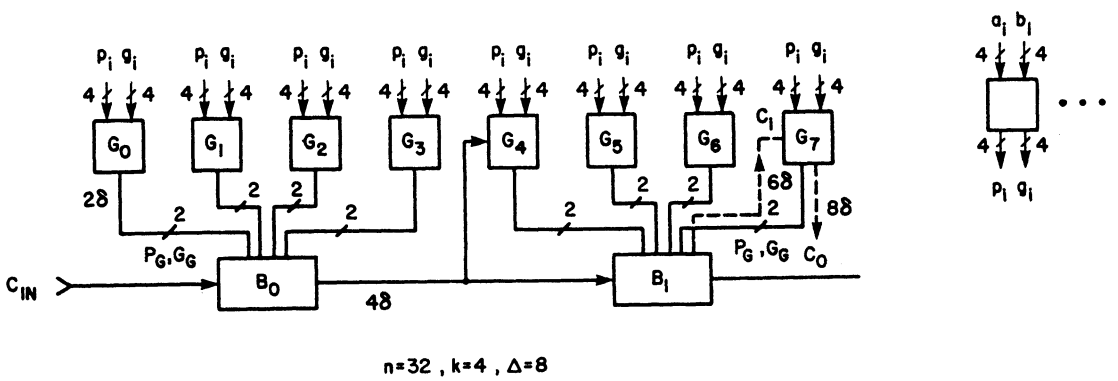


Fig.8. Carry Signal Delay of a CLA Adder of the size n=32 bits

n	$\Delta(CSA)$	$\Delta(CLA)$
16	5	6
32	7	8
48	9	10
64	10	10

Table 2. Delay comparison between CSA of Section 3. and CLA

In the first case (Table 1.) we notice that CLA is faster than our method of Section 1. resulting in equal delay for n=16 and ending up with 40% advantage for n=64. However, in the second case (Table 2.) our method of Section 3. shows advantage in speed over CLA of approximately 20% for n=16 and no advantage for n=64.

Our scheme exhibits regularity in fan-in and fan-out throughout the entire carry chain and therefore yields simpler and regular implementation. The amount of logic required to implement our scheme should therefore be smaller then that of CLA. Therefore its implementation in VLSI technology should yield even better results.

5. CONCLUSION

Our goal in implementing this scheme was to achieve a regular structure without using an excessive chip area. Comparison of various ALU implementation schemes for area and power as a parameter show that after some point small improvement in speed is achieved by a large investment in area or power [6],[7]. In our approach we argue that this incremental improvement in speed is diminished due to the overhead in the wiring capacitance and device size. Therefore our approach is to maximize the speed not by increasing the power or adding a substantial amount of logic but rather by optimizing on the path of the signal with the critical timing (carry signal) by designing the ALU around the carry path.

The method described in Sections 2. and 3. is especially applicable for the floating-point fraction ALU which is usually of large size n.

REFERENCES

[1] T.Kilburn, D.B.G.Edwards and D.Aspinall, *Parallel Addition in Digital Computers: A New Fast "Carry" Circuit,* Proc. IEE, Vol.106, Pt.B. P.464, September 1959.

[2] M.Lehman, *A Comparative Study of Propagation Speed-up Circuits in Binary Arithmetic Units*, Information Processing, 1962, Elsevier-North Holand, Amsterdam, 1963, p.671.

[3] M.Lehman and N.Burla, *Skip Techniques for High-Speed Carry-Propagation in Binary Arithmetic Units*, IRE Transactions on Electronic Computers, December 1961, p.691.

[4] L.P.Morgan and D.B.Jarvis, *Transistor Logic Using Current Switching and Routing Techniques and its Application to a Fast "Carry" Propagation Adder*, Proc. IEE, pt.B, Vol.106, p.467, September 1969.

[5] Kai Hwang , *Computer Arithmetic: Principles Architecture and Design*, John Wiley and Sons, 1979.

[6] S.Ong and D.E.Atkins, *A Comparison of ALU Structures for VLSI Technology*, Proc. of the 6th Symposium on Computer Arithmetic, June 20-22, 1983, Aarhus University, Aarhus, Denmark.

[7] Robert K. Montoye, P.W.Cook, *Automatically Generated Area, Power, and Delay Optimized ALUs*, IEEE ISSCC Digest of Technical Papers, February 23-24, 1983, New York.

[8] A.Bilgory and D.D.Gajski, *Automatic Generation of Cells for Recurrence Structures*, Proc. of 18th Design Automation Conference, Nashville, Tennessee 1981.

[9] V.F.Demyanov, V.N.Malozemov, *Vvedenie v Minimaks*, Izdatelstvo Nauka , Fiziko-Matematicheskoi Literaturi, Moskva 1972.

A 4.5 NS 96B CMOS Adder Design

AJAY NAINI, DAVID BEARDEN AND WILLIAM ANDERSON
MICROPROCESSOR AND MEMORY TECHNOLOGIES GROUP, MOTOROLA INC.
6501 WILLIAM CANON DRIVE WEST, AUSTIN, TEXAS 78735-8598

ABSTRACT

A new approach to the design of high-speed adders using the carry look-ahead principle is presented. These techniques discuss a new organization for the carry-chain, with minimal area impact, which minimizes the latency while maintaining modularity. Application of these techniques to a 96-bit adder, implemented in a CMOS process with 1.0 μm design rules, shows a critical path delay of 4.5 ns.

INTRODUCTION

Carry look-ahead (CLA) addition has become one of the more popular techniques for implementing fast adders because of the resulting speed, area, and modularity[1]. However, as the word size of the adders increase, the conventional carry-chain delay easily limits cycle time. In an attempt to decrease carry-chain delay, new circuit techniques have been applied to CLA addition[2]. This paper discusses a new organization for the carry-chain to minimize latency. As an example, a 96-bit adder design will be discussed.

The recursive method of CLA addition is well known. In general, fan-in limits carry look-ahead to groups of four bits. Because of this, multi-level look-ahead structures are used for larger words. Fig. 1a shows the structure for a 64-bit conventional CLA carry-chain in a group size of four, and the CLA equations are shown in Fig. 1b. Each carry-block generates three local carry-out terms from the P,G terms and the carry-in. Fig. 1c shows a circuit implemention of a 4-bit group PG-block and carry block. There are three levels of delay to the most significant carry-out term. The PG-block generates the 4-bit group P,G terms.

The critical path for the adder which is highlighted in Fig. 1a is:

$A,B - G_0 - G_{3:0} - G_{15:0} - C_{16} - C_{32} - C_{48} - C_{52} - C_{56} - C_{60} - C_{61} - C_{62} - C_{63} - S_{63}$

MODIFIED CLA STRUCTURE

Fig. 2a shows the modified CLA structure. The PG-block here generates intermediate group P terms ($P_{2:0}$, $P_{1:0}$) and group G terms ($G_{2:0}$, $G_{1:0}$), in addition to the 4-bit group P,G terms. A Manchester chain structure is used in the generation of the group G terms to reduce the transistor count. The carry-block generates three local carry-out terms (C_1, C_2, C_3). The associated circuit structure for the PG-block and carry block is shown in Fig. 2b. The generation of the intermediate group P,G terms means the delay to any carry-out term is only one level.

The critical path for the modified CLA structure which is highlighted in Fig. 2a is:

$A,B - G_0 - G_{3:0} - G_{15:0} - G_{47:0} - C_{48} - C_{60} - C_{63} - S_{63}$

Table 1 details the timing associated with each of the above critical paths. These timing numbers are based upon a 1 μm design rules CMOS process ($L_{eff} = 0.8$ μm), 5.0 V, 25° C. As can be seen, the 64-bit modified CLA addition is faster than the conventional 64-bit addition. The reason for the faster speed is the availability of the intermediate group P,G terms at every PG-block, which enabled the most significant carry-out term delay of every carry-block to be a one-level delay. The PG-block of the modified CLA design has 41 transistors as opposed to 19 transistors for the conventional CLA

Reprinted from *IEEE 1992 Custom Integrated Circuits Conference*, pp. 25.5.1-25.5.4, 1992.

design. However, this will minimally impact area because these transistors arise in the intermediate bits, which will not be the limiting factor in determining the bit cell size.

CARRY INPUT REQUIREMENTS

In the critical path for the conventional 64-bit CLA adder, the carry-in, C_0, is needed in the generation of C_{16} (Fig. 1a). In the modified 64-bit CLA adder, the carry-in is needed in the generation of C_{48} (Fig. 2a). This means the carry-in to the respective adders should be valid by 1.7 ns and 2.9 ns (Refer to Table 1 for timing numbers). This tolerance of a late carry-in of the modified adder may be applied in the construction of larger adders.

A 96-BIT ADDER USING MODIFIED CLA

The 96-bit adder shown in Fig. 3 consists of a high-order 64-bit modified adder and a low-order 32-bit conventional adder. The low-order 32-bit adder has the carry-out, C_{32}, valid by 2.35 ns (refer to Table 1 for timing numbers). Since this carry-out delay falls within the carry-in requirement (2.9 ns) of the 64-bit adder, the 32-bit adder timing does not affect the critical path. Therefore, the low-order 32-bit addition is effectively transparent to the operation of the higher-order 64-bit addition. This phenomenon enables the 96-bit addition to be performed in 4.5 ns, which is the same time as the modified 64-bit addition, and in a lesser time than the 64-bit conventional CLA addition.

REFERENCES

[1] S. Waser and M. Flynn, *Introduction to Arithmetic for Digital Systems Designers,* Holt, Rinehart, and Winstion, New York, 1982, pp. 83-88.

[2] I. Hwang and A. Fisher, "A 3.1 ns 32b CMOS Adder in Multiple Output Domino Logic," *ISSCC Digest of Technical Papers,* Feb. 1988, pp. 140-141.

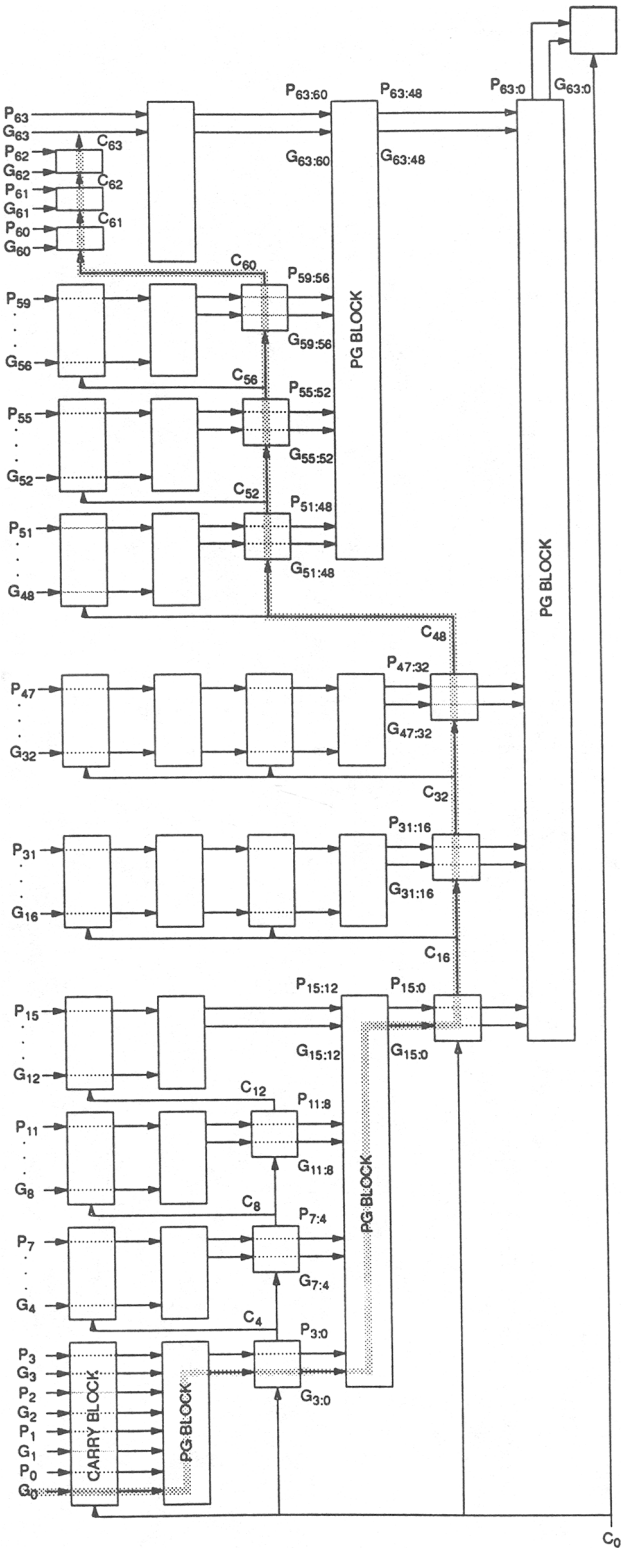


Figure 1a. Conventional CLA Structure

$P = A \oplus B$ (propogate)

$G = A \cdot B$ (generate)

C_0 (carry-in)

$C_1 = G_0 + C_0 \cdot P_0$

$C_2 = G_1 + C_1 \cdot P_1 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1$

$C_3 = G_2 + C_2 \cdot P_2 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2$

$C_4 = G_3 + C_3 \cdot P_3 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3$

$= G_{3:0} + C_0 \cdot P_{3:0}$

$C_8 = G_{7:4} + C_4 \cdot P_{7:4} = G_{7:4} + G_{3:0} \cdot P_{7:4} + C_0 \cdot P_{3:0} \cdot P_{7:4}$

...

...

$C_{16} = G_{15:0} + C_0 \cdot P_{15:0}$

$C_{64} = G_{63:0} + C_0 \cdot P_{63:0}$

$S_n = P_n \oplus C_n$

Figure 1b. Carry-Lookahead Equations

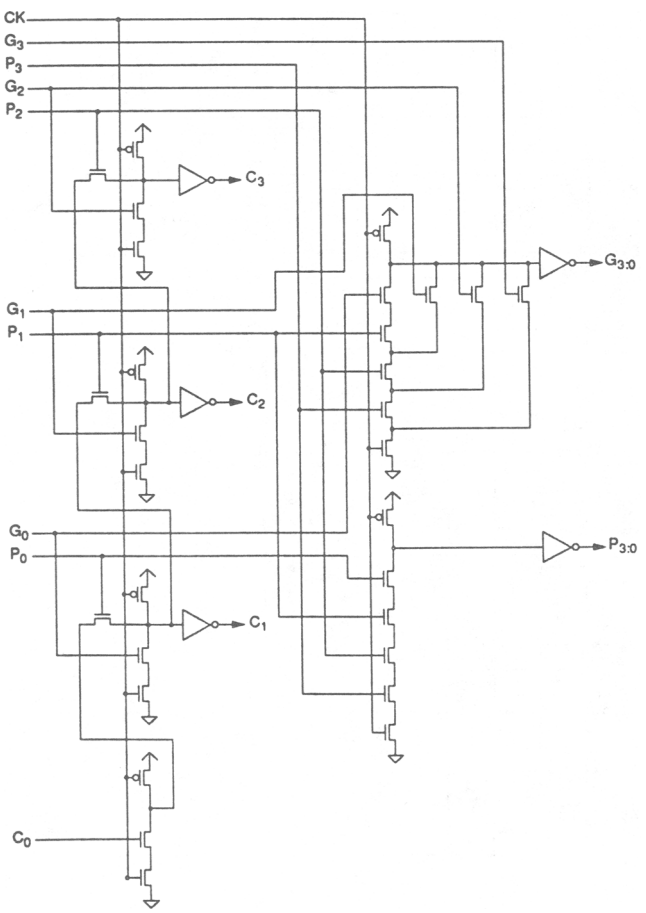


Figure 1c. Circuit Implementation of Conventional 4b Group PG and Carry Blocks

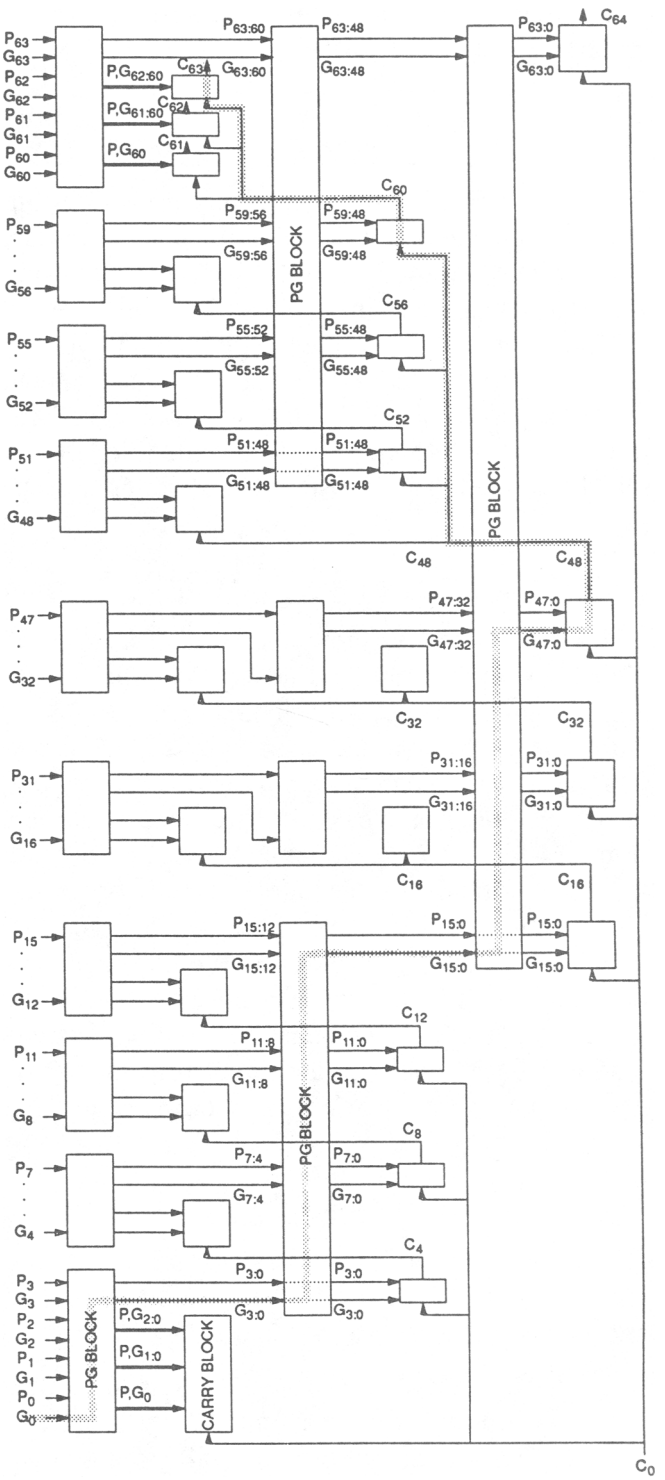


Figure 2a. Modified CLA Structure

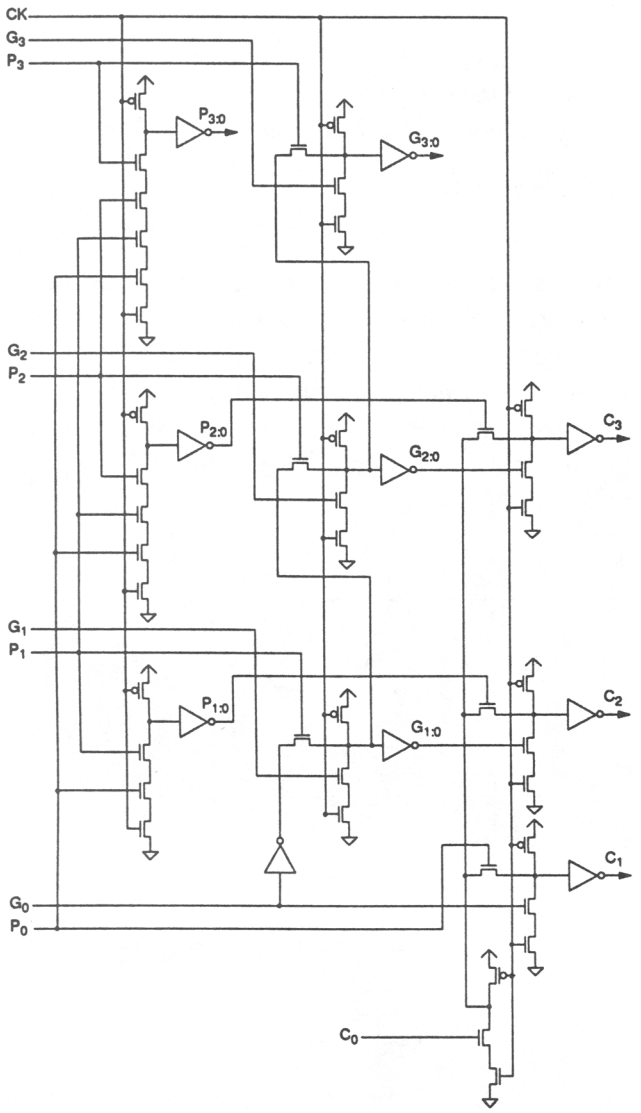


Figure 2b. Circuit Implementation of Modified 4b Group PG and Carry Blocks

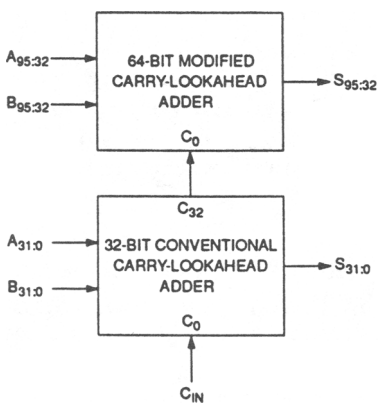


Figure 3. 96-Bit Adder

CONVENTIONAL CLA ADDER (FIGURE 1A)	CRITICAL PATH	$A,B - G_0$	$G_0 - G_{3:0}$	$G_{3:0} - G_{15:0}$	$G_{15:0} - C_{16}$	$C_{16} - C_{32}$	$C_{32} - C_{48}$	$C_{48} - C_{52}$	$C_{52} - C_{56}$	$C_{56} - C_{60}$	$C_{60} - C_{61}$	$C_{61} - C_{62}$	$C_{62} - C_{63}$	$C_{63} - S_{63}$	TOTAL = 5.4NS				
	DELAY (NS)	0.4	0.65	0.65	0.3	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.6					
MODIFIED CLA ADDER (FIGURE 2A)	CRITICAL PATH	$A,B - G_0$	$G_0 - G_{3:0}$	$G_{3:0} - G_{15:0}$	$G_{15:0} - G_{47:0}$	$G_{47:0} - C_{48}$	$C_{48} - C_{60}$	$C_{60} - C_{63}$	$C_{63} - S_{63}$	TOTAL = 4.5NS									
	DELAY (NS)	0.4	0.9	0.9	0.7	0.3	0.35	0.35	0.6										

Table 1. Timing Characteristics for 1.0 µm CMOS process with VDD = 5V and 25° C

Improved CLA Scheme with Optimized Delay

BRIAN D. LEE AND VOJIN G. OKLOBDZIJA*

Department of Electrical Engineering and Computer Science, University of California, Berkeley

Received January 31, 1991; Revised April 25, 1991.

Abstract. The delay characteristics of carry-lookahead (CLA) adders are examined with respect to a delay model that accounts for fan-in and fan-out dependencies. Though CLA structures are considered among the fastest topologies for performing addition, they have also been characterized as providing marginal speed improvement for the amount of hardware invested. This analysis shows that this inefficiency can be explained by the suboptimal nature of common CLA implementations. Simulation results show that the CLA structures in wide use can be improved by varying the block sizes and the number of levels within each adder. Examples of optimal CLA structures are given and heuristic methods for finding these structures are presented.

1. Introduction

Analysis of carry-lookahead adders is important in the design of high performance machines. In these designs, processor speed is a primary concern and carry-lookahead structures are often used because their delay times exhibit log dependence on the size of the adder and they are considered among the fastest circuit topologies for performing addition. However, adder comparisons [1], [2] have ranked CLA structures low on effective hardware utilization and this apparent inefficiency raises concerns over the optimality of current CLA implementations. Simulation results from this research show that the commonly used CLA structures can be improved by varying block sizes and levels within the adder.

Typical CLA implementations are made of lookahead units of relatively fixed sizes. This artificial constraint produces slack in the circuit and results in poor hardware utilization. The strategy of varying group sizes to reduce slack and improve performance is a promising idea and has been used successfully on carry-skip adders [3], [4]. A natural extension of this method is to also vary the number of lookahead levels [5], [6]. The example structures in figure 1 illustrate what it means to vary group sizes and lookahead levels. Each box corresponds to a BCLA/CLA unit and the enclosed number gives the unit size in bits. The dotted lines in figure 1(d) represent a connection to a primary input of the CLA network. All other connections to primary inputs are implicit and not shown. Since an accurate measure of the available slack is required to effectively implement these strategies, this work uses a delay model that accounts for fan-in and fan-out dependencies. The parameter values used in the model are based on industrial data.

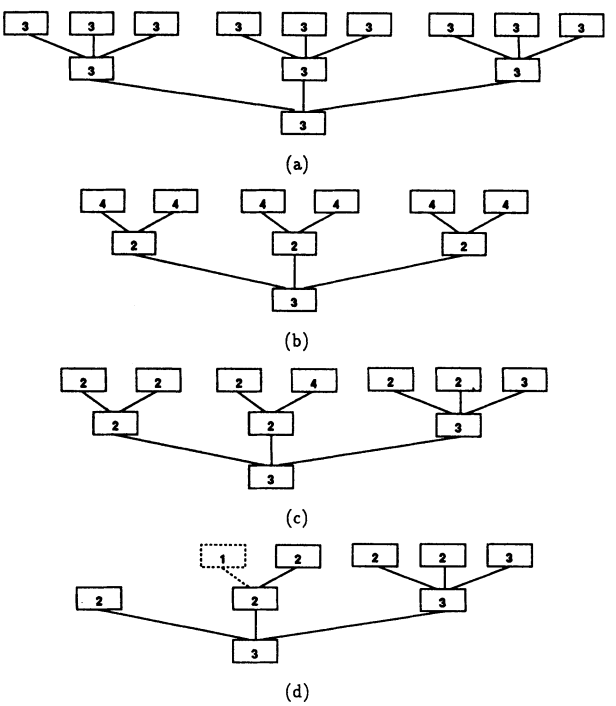


Fig. 1. Degrees of freedom in group sizes and lookahead levels: (a) fixed groups and fixed levels, (b) variable-sized groups between levels and fixed levels, (c) variable-sized groups anywhere (between levels and within levels) and fixed levels, and (d) variable-sized groups anywhere and variable levels.

*Currently with IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

A simulation program has been written to compare different CLA structures. Preliminary data on varying block sizes was obtained through exhaustive search. Based on this data and an analysis of delay and slack in the CLA scheme, heuristics were chosen to find structures with completely variable block sizes and levels. The structures found by these heuristics are faster than more constrained topologies.

2. Carry Lookahead Structure

The simulation results in this paper are based on carry-lookahead adders that implement full lookahead. A description of the basic organization of carry-lookahead adders can be found in references such as [7]. Each adder consists of three main components—the propagate and generate generation circuitry, the carry-lookahead network, and the sum generation circuitry. This work concerns varying the sizes of circuit blocks and the number of levels in the carry-lookahead network to optimize adder delay.

Given an n-bit adder with inputs, A and B, the logic equations for producing the initial propagate and generate signals and the final sum signals are

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

and

$$S_i = P_i \oplus C_{i-1}$$

for $0 \le i \le n - 1$. The simulations assume that P_i and S_i are produced by monolithic XOR gates instead of two levels of NAND gates and the sum XOR gate is assumed to have a fan-out of one.

The carry-lookahead network in a full carry-lookahead adder consists of a tree of block carry-lookahead (BCLA) units rooted at a single carry-lookahead (CLA) unit. Two different implementations of BCLA/CLA units are analyzed. Their performance differences are discussed later.

The first implementation generates carry signals in two levels of logic. Four-bit versions of these circuits are shown in figures 2 and 3. A k-bit carry-lookahead unit of this type generates

$$C_j = G_j + G_{j-1}P_j + G_{j-2}P_{j-1}P_j + \ldots + G_0P_1 \ldots P_j + C_{-1}P_0 \ldots P_j$$

where $0 \le j \le k - 1$ and a k-bit block carry-lookahead unit of this type generates

$$P^* = P_0P_1 \ldots P_{k-1}$$

$$G^* = G_{k-1} + G_{k-2}P_{k-1} + \ldots + G_0P_1 \ldots P_{k-1}$$

$$C_j = G_j + G_{j-1}P_j + G_{j-2}P_{j-1}P_j + \ldots + G_0P_1 \ldots P_j + C_{-1}P_0 \ldots P_j$$

where $0 \le j \le k - 2$.

The second implementation generates carry signals in three levels of logic. Four-bit versions of these circuits are shown in figures 4 and 5. A k-bit carry-lookahead unit of this type generates

$$C_j = G_j^* + C_{-1}P_j^*$$

where $0 \le j \le k - 1$ and a k-bit block carry-lookahead unit of this type generates

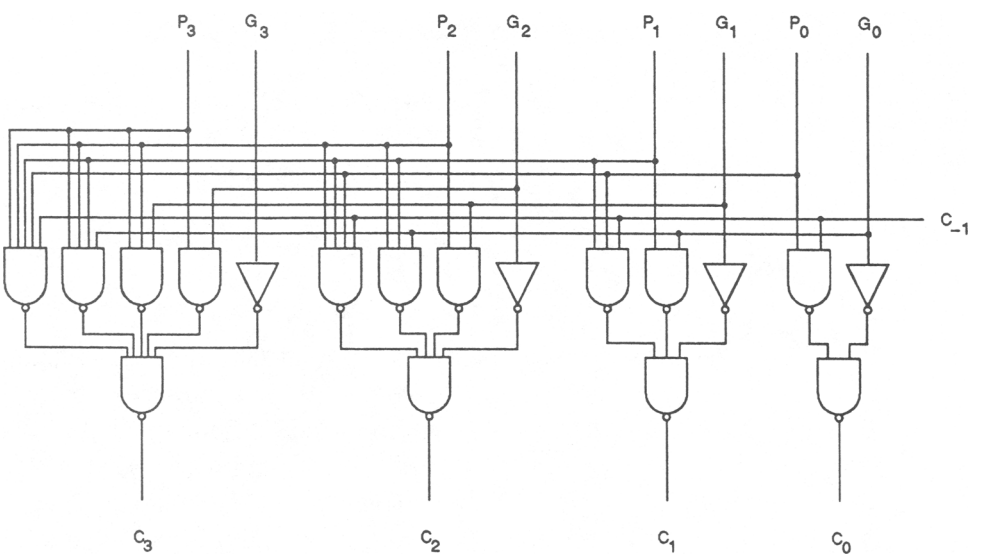


Fig. 2. A 4-bit carry-lookahead unit with 2-level C_i logic.

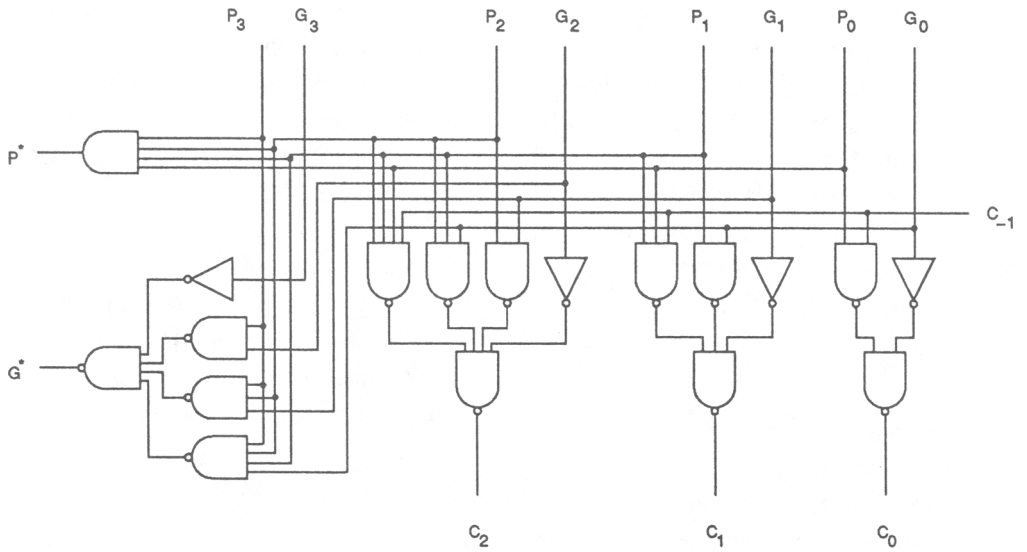


Fig. 3. A 4-bit block carry-lookahead with 2-level C_i logic.

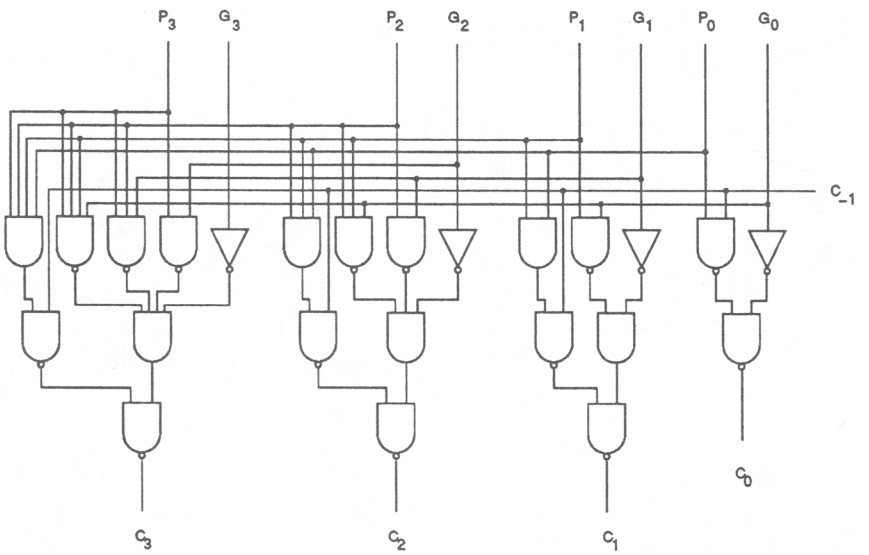


Fig. 4. A 4-bit carry-lookahead unit with 3-level C_i logic.

$$P^* = P^*_{k-1}$$
$$G^* = G^*_{k-1}$$
$$C_j = G^*_j + C_{-1}P^*_j$$

where $0 \leq j \leq k - 2$. For both circuit blocks,

$$G^*_j = G_j + G_{j-1}P_j + G_{j-2}P_{j-1}P_j + \ldots + G_0P_1 \ldots P_j$$

and

$$P^*_j = P_0P_1 \ldots P_j.$$

The simulations also assume that C_{-1} and all A_i, B_i are latched and available at time $t = 0$. Fan-out loading of A_i, B_i and C_{-1} is ignored and the adder delay is calculated as the time required to generate the slowest signal from among S_i and C_{n-1}.

3. Carry Lookahead Optimization

The basic goal of this research is to show how to optimize CLA structures by varying group sizes and lookahead levels. The purpose of these operations is to exploit the delay differences that represent under-utilized time. Early signals can be delayed by modifying the network to make the signals the result of more lookahead computation. Since this allows the addition of larger operands in the same amount of time, slack

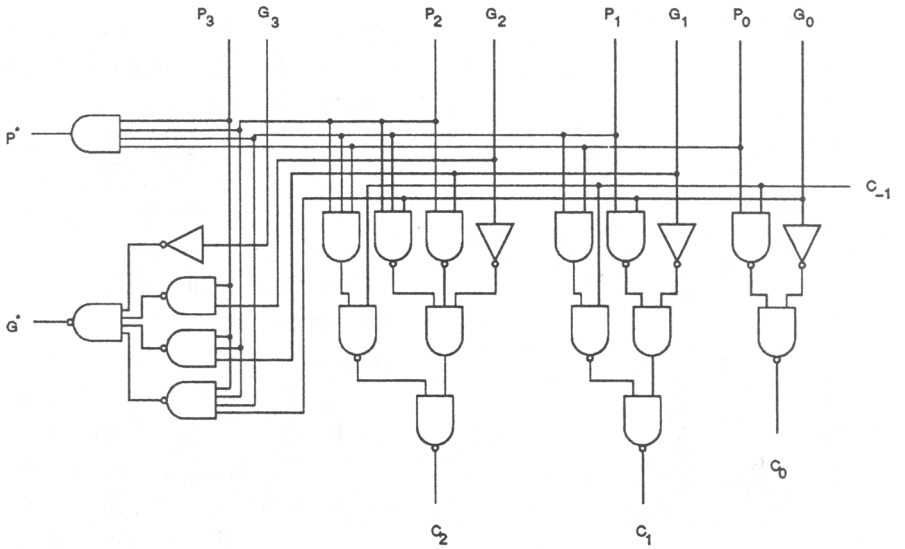


Fig. 5. A 4-bit block carry-lookahead unit with 3-level C_i logic.

reduction corresponds to adder structure optimization. The optimization requires a delay model that accurately measures slack in the circuit, an examination of the delay characteristics of the circuit blocks, and an analysis of critical delay paths to identify slack in the circuit. These requirements lead to a heuristic method for determining optimal CLA structures.

3.1. Delay Model

Logic gate delays are modeled as

$$\text{delay} = f(\text{fi}, \text{fo})$$

$$= A + B \cdot \text{fi} + (D + E \cdot \text{fi}) \cdot \text{fo}.$$

Simpler delay models that use unit gate delays are inadequate because they do not reveal all the slack in the circuit.

The simulations use the following delay functions:

$$t_{NAND} = 0.1058 + 0.1175\text{fi} + (0.0825 + 0.0148\text{fi})\text{fo}$$

$$t_{AND} = 0.2825 + 0.1675\text{fi} + (0.0911 + 0.0037\text{fi})\text{fo}$$

$$t_{INV} = 0.265 + 0.1016\text{fo}$$

$$t_{XOR} = 0.945 + 0.05645\text{fo}.$$

The constants in these functions are based on LSI Logic Corporation's 1.5 Micron Compacted Array™ Technology [8]. To limit complexity, the models assume that that all logic gates are single, possibly large, gate structures rather than multiple levels of smaller gates.

3.2. BCLA/CLA Delay Characteristics

An understanding of BCLA/CLA delay characteristics is important for finding opportunities to reduce slack in CLA structures. In particular, the fan-in and fan-out properties of the circuits must be analyzed.

The input and output loading on the propagate and generate signals of a BCLA can be derived by induction on the circuits of figures 2, 3, 4, and 5. By inspection, the following results are obtained. Given a k-bit BCLA unit connected to the jth input of an m-bit BCLA unit

- G^* of the k-bit BCLA unit has fan-out $(m - j)$.
- G^* of the k-bit BCLA unit has worst case fan-in k. Specifically, in the two-level implementation of G^*, the first level NAND gate i associated with input G_i has fan-in $k - i$.
- P^* of the k-bit BCLA unit has fan-in k.
- P^* of the k-bit BCLA unit has fan-out $(m - j)(j + 1)$

Examples of these relationships are shown graphically in figures 6 and 7. The analysis for a BCLA unit feeding into a CLA unit is similar and gives the same results.

The loading on carry signals may be derived by a similar analysis of the circuit diagrams. Each carry signal, C_i, of a BCLA/CLA unit is the C_{-1} of BCLA units on previous levels. An example of this is shown in figure 8. In particular, it connects to its $(i + 1)th$ fan-in unit, the zeroth fan-in unit of its $(i + 1)th$ fan-in unit, the zeroth fan-in unit of the zeroth fan-in unit of its $(i + 1)th$ fan-in unit, etc. Each carry signal also connects to an XOR gate in the sum generation circuitry. A BCLA unit of size k contributes $k - 1$ to the fan-out loading of its input C_{-1} signal. In the two-level implementation

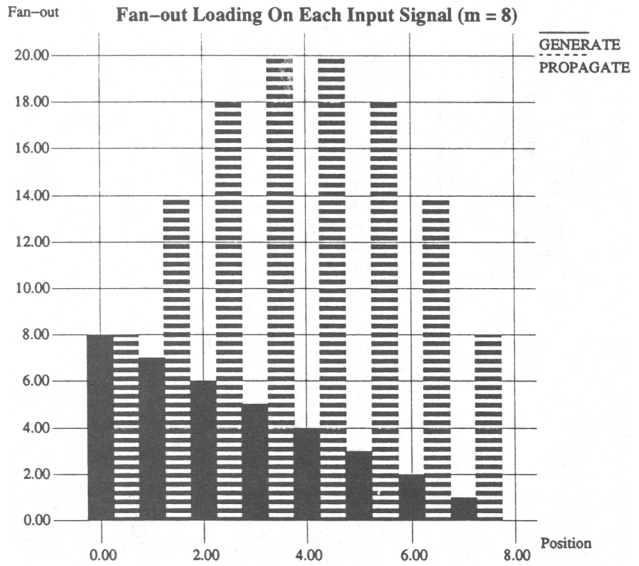


Fig. 6. Fan-out loading on input signals of an 8-bit BCLA/CLA unit.

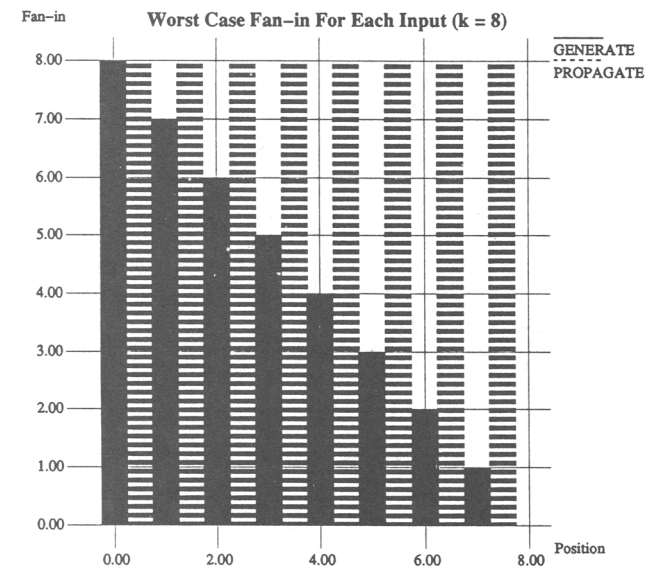


Fig. 7. Worst case fan-in gates for the inputs of an 8-bit BCLA unit.

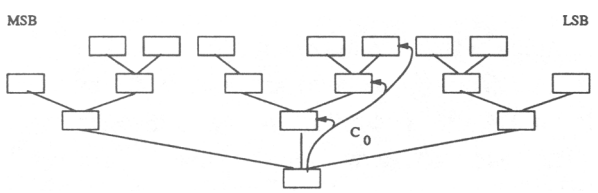


Fig. 8. Fan-out path within the CLA network of C_0 of the final lookahead unit.

of C_i, the worst case first-level NAND gate has fan-in $i + 2$. In the three-level implementation of C_i, C_{-1} always connects to an input NAND gate with a fan-in of 2.

These fan-in and fan-out properties have an important effect on the critical delay analysis in BCLA/CLA units. Unlike typical analyses, the P^* delay cannot be neglected with respect to the G^* delay. Even though computing P^* requires one less level of logic than calculating G^*, the potentially high group propagate fan-out loading may place generation of P^* on the critical path. The P^* delay path should be compared to the G_0-to-G^* delay path which contains the worst case fan-in gate.

Another important delay path is the carry propagation (assimilation) path. This is the path from C_{-1} to some C_i and is critical when C_{-1} arrives much later than all P_i and G_i. Assuming a k-bit BCLA unit, the worst case path for the two-level implementation of C_i is from C_{-1} to C_{k-2}. For the corresponding three-level implementation of C_i, the paths are equal for all i because C_{-1} feeds gates of constant fan-in. When C_{-1}-to-C_i delays are a significant fraction of the total adder delay, the three-level implementation should produce faster adders than the two-level implementation because of its superior fan-in properties. This condition should hold for larger adders.

An issue related to BCLA delay is the relationship between group size and number of lookahead levels. Clearly, larger BCLA units have longer delays than smaller units. Also, adding more levels of BCLA units tends to increase delay because of the extra logic levels. However, at some size, implementing a single large BCLA unit as multiple levels of BCLA units is advantageous. Unfortunately, determining this breakpoint is difficult because the delay of a BCLA unit depends on the block sizes on the next level of lookahead and typically, those sizes are determined by the number and sizes of blocks on all previous levels.

3.3. Critical Path Analysis

Identifying critical paths in CLA structures is important because the remaining noncritical delay paths represent opportunities for slack reduction. Standard CLA analyses assume that the critical path in the adder is always as shown in figure 9. Unfortunately, the validity of this assumption is not guaranteed when variable group sizes and lookahead levels are allowed. However, the actual critical path will have an analogous form. The first part of the critical path is the delay to the generation of a carry signal in the CLA unit and the last part of the critical path is the propagation of this carry signal back through some subtree of the BCLA network. Furthermore, each subtree of the network has an analogous critical path. Opportunities to reduce slack can be found in each portion.

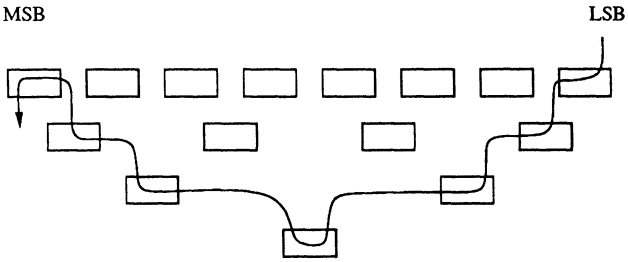


Fig. 9. Critical path assumed by typical analyses.

The delay to the CLA unit depends only on G^* and P^* delays. Most of this delay is expected to be from G_0-to-G^* delays because this path has both high fan-out loading and worst case fan-in gates. On a given level, the critical path might depend on the generation of P^*, but on the next level, the critical path will most likely be the group generate computation because G^* is a function of all the input propagate signals except P_0. The adder delay after the CLA unit depends only on carry propagation delays. All P_i and G_i signals have already settled and the critical path follows the worst C_{-1}-to-C_i path in a BCLA unit on each level.

Assuming that the generation of G^* is the critical path through a BCLA unit, then the group feeding G_i can be larger than the group feeding G_j for $i > j$, since G_j feeds a higher fan-in gate. This will reduce the slack between the different G_i-to-G^* delays. This argues for fewer levels of smaller lookahead units in the least significant bits than in the most significant bits of every subtree of lookahead units.

The slack in the second portion of a critical path arises from different C_i delay times in a lookahead unit. The subtrees fed by C_j should be faster than those fed by C_i for $j > i$. This indicates that fewer levels of smaller lookahead units should handle the most significant bits of each subtree than should handle the least significant bits.

The criteria for each portion of the critical path contradict each other. This indicates that the best opportunities to increase group sizes or add levels of lookahead may occur in the middle bits of each subtree rather than at the ends. Unfortunately, this criteria cannot be used to make specific decisions except for trivial cases. The problem is that the fan-in and fan-out dependencies of the delay model interfere with local optimizations. The whole adder must be analyzed because decisions on each level depend on the sizes of the next level which depend on the structure of the previous levels. A dynamic programming method is used to avoid the complexity of analyzing size combinations for entire adder structures.

3.4. Heuristics

The basic algorithm for finding an optimal CLA structure is based on dynamic programming. The generic pseudo-code for this approach is shown in figure 10. The problem of finding the optimal n-bit adder structure is reduced to a series of subproblems. Each subproblem requires finding an optimal $(k + b)$-bit adder structure given a k-bit adder structure. Note that the number of subproblems is bounded by $n - k_0$. The main components of the algorithm are the transformation step and the choice of initial adder structure.

The transformation step increases adder size by adding levels and/or increasing group sizes. Greedy heuristics are used to determine the location and magnitude of these increments. When increasing the size of a group, the extra bits are added to the most significant end of the group and the inputs to these new positions are connected directly to the initial propagate and generate generation circuitry. The solution of the transformation problem is simplified by formulating the decision to increase the number of lookahead levels as a decision to increase group sizes. This reduction is obtained by viewing all the inputs into the carry-lookahead network as groups of size one. Thus, increasing the size of a 1-bit *input* group often corresponds to increasing the number of lookahead unit levels in the adder. Increasing the size of such an input group does not always increase the number of levels because for any given network, multiple input groups may exist which will increase the number of lookahead levels and only the first one of these groups which is enlarged will actually increase the number of levels.

The choice for the initial adder structure is constrained to single CLA units. Ideally, the initial structure should be optimal and should admit a transformation path to a final optimal structure. The results show that the simple constraint of starting with a single CLA unit is effective.

Many different heuristics can be used to perform the transformation step and to choose the initial CLA unit size. Two different sets of heuristics were used to implement two different versions of the basic algorithm.

```
let i = 0
let initial structure = k0-bit adder

while ( ki ≤ n )
   transform the current ki-bit adder into a (ki + bi)-bit adder, bi ≥ 1
   let ki+1 = ki + bi
   let i = i + 1
end
```

Fig. 10. Basic algorithm for finding the structure of an n-bit adder.

The first version runs the basic algorithm for different starting CLA unit sizes and chooses the best structure from among all the runs. The current range of starting sizes is from 2 to 16. The transformation step constrains $b_i = 1$, $\forall i$. Each block in the network except for the CLA unit is considered for receiving this extra bit. For each block, the delay of the structure that results from increasing the block size by one is calculated. The transformation is performed by enlarging the block that results in the adder structure with the shortest delay.

The second version always starts with a 2-bit CLA unit and then uses a more complex transformation step. The basic goal of the transformation heuristic is to achieve the maximum increase in adder size per unit delay increase. Increasing by one the size of some group in an adder of size k and delay d results in an adder of size $k + 1$ and delay $d + \delta$. Depending on the location of the enlarged block, the size of other groups may also be increased without further increase in adder delay. In general, it is possible to increase the size of other groups to produce an adder of size $k + b$ and delay $d + \delta$ where $b \geq 1$. Given the selection of the initial enlarged group, the location of the remaining $b - 1$ bits can be found by adding as many extra bits as possible to each group of the network such that the adder delay remains $d + \delta$. The value of $b - 1$ is maximized by processing groups by level, starting with the final CLA unit and working back to the initial propagate and generate circuitry. Within levels, groups are processed from least significant to most significant. The transformation heuristic at step i calculates for each group j in the network

$$\text{benefit}_j = \frac{\min(b_j,\ n - k_i)}{\delta_j}$$

where

δ_j = increase in delay caused by increasing the size of group j by 1

b_j = maximum increase in adder size possible given the selection of group j for the initial 1 bit increment and a delay increase limit of δ_j

k_i = size of adder before transformation step i

and

n = desired final adder size.

The group j with the largest benefit value is chosen to transform the k_i-bit adder into a $(k_i + \min(b_j, n - k_i)$-bit adder.

A similar method has been implemented by [5]. That method also uses dynamic programming and the recursive step increases adder size by determining an optimal subtree of BCLA units to connect to a particular input position of the final CLA (BCLA) unit. Input positions are processed from least significant to most significant. In contrast, the method described here increases adder size by increasing group sizes or levels anywhere in the transitive fan-in of the final CLA unit. Also, the method of [5] requires delay models that are linear monotone nondecreasing non-negative functions of fan-in and fan-out and only optimizes the carry-lookahead network. This work allows delay models that are any arbitrary function of fan-in and fan-out and optimizes the whole adder. Simulating the whole adder exploits the delay differences between the initial propagate and generate signals and properly accounts for loading by the sum generation circuitry. Some comparisons with the results of the method in [5] are given in the next section.

4. Results

Figures 11 and 12 show results for the two different implementations of BCLA/CLA units. Each graph has three curves corresponding to requiring fixed-sized groups and fixed number of levels (**Fixed**), allowing different sized groups only on different levels and requiring fixed levels (**Inter-level**), and allowing variable sizes and levels anywhere (**Variable**). The delay values are for the best structures of each category. The **Fixed** and **Inter-level** curves were obtained by simulation of all possible combinations. If an integral number of groups of the chosen size for a level handled more

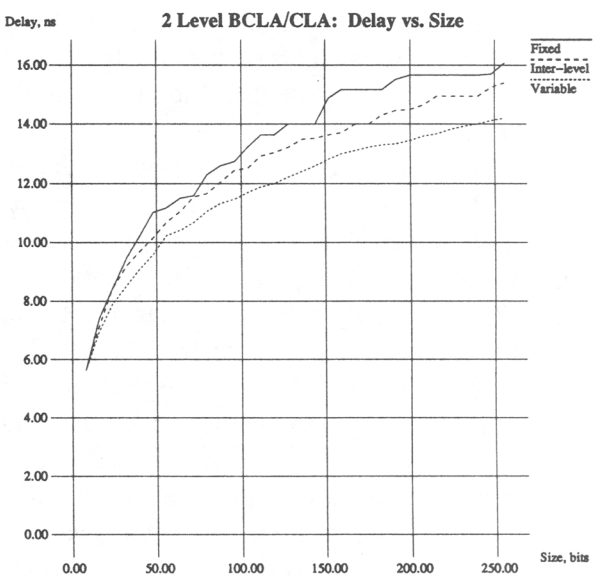


Fig. 11. Delay versus adder size in adders using a 2-level BCLA/CLA carry implementation.

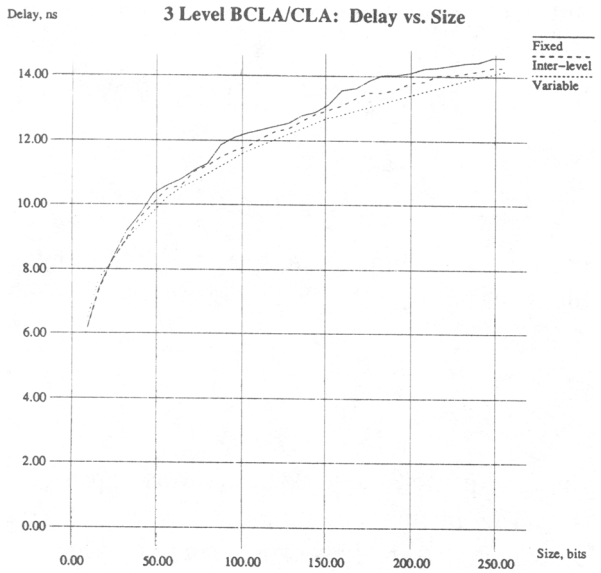


Fig. 12. Delay versus adder size in adders using a 3-level BCLA/CLA carry implementations.

signals on that level than was necessary, then the size of the group corresponding to the most significant bits of the adder was decreased to eliminate the excess capability. The **Variable** curve for the two-level BCLA/CLA carry implementation (figure 11) was obtained using the first set of heuristics. This version of the algorithm worked well for this lookahead implementation but not as well for the three-level implementation. The **Variable** curve for the three-level BCLA/CLA carry implementation (figure 12) was obtained using the second set of heuristics since it produced better results than the first set.

Figure 13 gives the optimal 32-bit adder structure found by each algorithm version. The 1's represent input from the initial bit positions.

The results correspond well to theory. Adders of fixed-size groups are slower than adders allowing variable inter-level groups. Adders with variable groups and levels are faster than both other types. The optimal **Variable** structures tend to have more levels and larger group sizes in the middle of the adder than on the ends. Results of comparing the different lookahead implementations are mixed. The **Fixed** and **Inter-level** three-level implementations performed much better than the two-level implementations. However, the heuristics improved the performance of the two-level implementations more than they improved the three-level implementations. The delay differences between the **Variable** adders of the two implementations is smaller, though the three-level implementation is still faster for larger adders.

Table 1 compares the delays of adders generated from the carry-lookahead network configurations of [5] with the delay adder structures found by the first version of the basic algorithm. The comparisons are based on the two-level implementations of the BCLA/CLA units. The delay models are those used in [5] plus an equivalent XOR delay model. The delay functions used are:

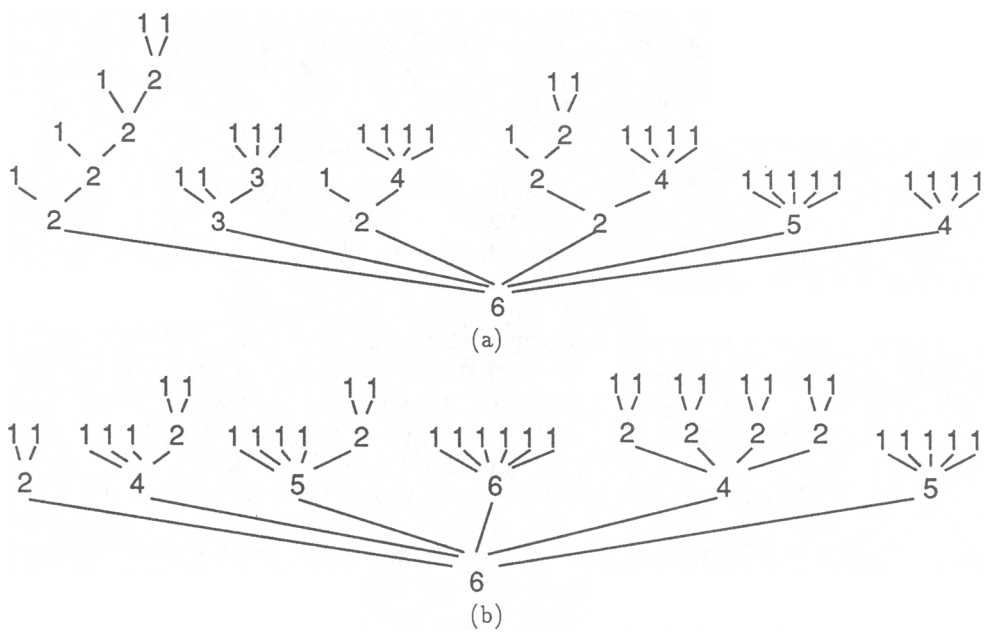


Fig. 13. 32-bit adders found by (a) the first set of heuristics using 2-level BCLA/CLA units (delay = 8.5 ns) and (b) the second set of heuristics using 3-level BCLA/CLA units (delay = 8.9 ns).

$$t_{NAND} = 5\text{fi} + 20\text{fo}$$

$$t_{AND} = 17 + 20\text{fi} + 5\text{fo}$$

$$t_{INV} = 12 + 5\text{fo}$$

$$t_{XOR} = 5\text{fo}.$$

The adder structures found by the first version of the basic algorithm are faster in all cases.

Table 1. Adder *sum* delays for configurations from [1] and for adders found by the first version of the basic algorithm.

	Delay	
Adder Size, bits	Chan, et al.	First Algorithm
8	407	394
16	524	514
24	627	584
32	652	637
48	736	721
64	821	779
66	824	779
84	877	834

Table 2 compares the adder structures corresponding to the delays in table 1. The structures are represented in parentheses notation as in [5]. For example, the adder structures shown in figures 13(a) and 13(b) are represented as

((1 (1 (1 (1 2)))) (1 1 3) (1 4) ((1 2) 4) 5 4)

and

(2 (1 1 1 2) (1 1 1 1 2) 6 (2 2 2 2) 5),

respectively. The numbers in the expression represent the block sizes at the highest level.

5. Conclusion

Varying group sizes and lookahead levels improves the performance of commonly used CLA implementations. Unfortunately, finding these improved structures is difficult because of delay fan-in and fan-out dependencies. In general, the whole adder structure must be known before a decision to increase group sizes or the number of lookahead levels can be made. Fortunately, simple heuristics can deal effectively with this problem. Simulation results show that heuristic methods can find CLA adder structures with variable group sizes and levels that are faster than more constrained carry-lookahead adders.

Table 2. Adder structures corresponding to the delay values in table 1.

Adder size, bits	Source	Adder Structure
8	Chan, et al.	(1 1 2 2 2)
	First Algorithm	(1 2 3 2)
16	Chan, et al.	(((1 2) (1 1 2) 4 (2 1)) 2)
	First Algorithm	((1 (1 2)) (1 1 2) 4 (2 1 1))
24	Chan, et al.	((1 3) (2 (2 2)) (2 3 2) (3 2 2))
	First Algorithm	(1 (1 (1 2)) (1 1 2) 4 4 (2 1 1) 3)
32	Chan, et al.	(((1 (1 (1 2))) ((1 2) ((1 2) 3)) (2 3 2) (3 2 2)) 2 2)
	First Algorithm	(((1 (1 2)) (1 1 3) (1 1 2 2) (2 3 2) (3 2 2)) 2 1)
48	Chan, et al.	(((1 2) ((1 (1 2) (2 2)) ((1 2) (1 2) (3 2)) ((1 2) 3 3 2) (4 3 3))) 3 2)
	First Algorithm	(((1 (1 (1 2))) (1 1 (1 1 2)) (1 1 (1 2) 4) ((1 2) 3 (2 1 1)) (4 (2 1 1) 3)) (2 1 1) 3)
64	Chan, et al.	((2 3 ((1 2 3 3) (2 3 4 (2 2)) ((1 2 2) (2 2 2) (3 2)) ((2 2 2) (3 2) 3)) (3 2)) 2)
	First Algorithm	(((1 (1 (1 (1 2)))) (1 1 (1 1 3)) (1 1 (1 2) 4) ((1 (1 2)) (1 1 2) (2 2 2)) ((1 1 2 1 1) (3 2 2) (3 2 1))) (2 2 1) (2 1 1))
66	Chan, et al.	((2 3 ((1 2 3 3) (2 3 4 (2 2)) ((1 2 2) (2 2 2) (3 2)) ((3 2 2) (3 2) (2 2))) (3 2)) 2)
	First Algorithm	(((1 (1 (1 (1 2)))) (1 1 (1 1 3)) (1 1 (1 2) 4) ((1 (1 2)) (1 1 3) (2 3 2)) ((1 1 2 1 1) (3 2 2) (3 2 1))) (2 2 1) (2 1 1))
84	Chan, et al.	((3 4 ((2 3 (1 2 2) (3 2 2)) ((1 2 2) (2 3 2) (3 2 2)) ((2 3 2) (3 2 2) (3 2)) ((3 2 2) (3 2) 3)) (3 2)) 2)
	First Algorithm	(((1 (1 (1 (1 2)))) (1 1 (1 1 (1 1 2))) (1 1 (1 (1 2)) (1 1 2 2)) ((1 (1 2)) (1 1 3) ((1 2) 3 3)) ((1 1 2 2.2) (3 3 3) ((2 1 1) 3 2))) (3 3 2) (3 2 1))

Work is in progress to re-run the simulations in this paper for ECL delay data. This is particularly important in the domain of high performance machines where bipolar is the dominant technology. Also, more heuristics and implementations will be examined. For example, a hybrid of the two sets of heuristics presented here will be tried and an adder structure that uses the three-level BCLA carry implementation with the two-level CLA carry implementation will be tested.

Acknowledgments

The authors would like to thank Professor Pak Chan for sharing his ideas and insight on carry-lookahead adders. His many helpful discussions and suggestions are greatly appreciated.

References

1. R. Sherburne, Jr., "Processor design tradeoffs in VLSI," Technical Report UCB/CSD 84/173, University of California, Berkeley, 1984.
2. J. Sklansky, "An evaluation of several two-summand binary adders," *IRE Trans.*, EC-9(2), 1960, pp. 213–226.
3. V.G. Oklobdzija and E.R. Barnes, "On implementing addition in VLSI technology," *Journal of Parallel and Distributed Computing*, 5, 1988, pp. 716–728.
4. S. Turrini, "Optimal group distribution in carry-skip adders," In *Proceedings of the 9th Symposium on Computer Arithmetic*, 1989, pp. 1–18.
5. P.K. Chan, M.D.F. Schlag, C.D. Thomborson, and V.G. Oklobdzija, "Delay optimization of carry-skip adders and block carry-lookahead adders," in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, June 1991, pp. 159–164.
6. B.D. Lee and V.G. Oklobdzija, "Optimization and speed improvement analysis of carry-lookahead adder structure," In *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 1990, pp. 918–922.
7. K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, New York: Wiley & Sons, 1979.
8. LSI Logic Corporation, *Databook: 1.5 Micron Compacted Array™ Technology*, 1987.

High-Speed Binary Adder

HUEY LING

Based on the bit pair (a_i, b_i) truth table, the carry propagate p_i and carry generate g_i have dominated the carry-look-ahead formation process for more than two decades. This paper presents a new scheme in which the new carry propagation is examined by including the neighboring pairs (a_i, b_i; a_{i+1}, b_{i+1}). This scheme not only reduces the component count in design, but also requires fewer logic levels in adder implementation. In addition, this new algorithm offers an astonishingly uniform loading in fan-in and fan-out nesting.

Introduction

The traditional recursive formula for carry propagation has dominated the carry handling process in the computer industry for more than two decades. Today, adder designs based on a similar technique include Amdahl V6, IBM 168, and IBM 3033.

The recursive formulation of carry is based on the bit pair (a_i, b_i) truth table. By examining the local bit pair, carry propagate p_i and carry generate g_i are formed. The high-order carries are generated by nesting the p_i and g_i together. By considering the adjacent bit pairs (a_i, b_i; a_{i+1}, b_{i+1}), a new recursive formula is obtained for new carry propagation. The comparison between this new scheme and the existing scheme will be discussed in the following sections. The detailed implementation, circuits, and logic level count are also included. Surprisingly, this method offers an astonishingly uniform loading in fan-in/fan-out nesting.

The formation of new carry and sum

This paper introduces a new approach to represent the new carry formation and propagation based on the concept of the complementing signal which was introduced in 1965 [1]. To examine the impact of this complementing signal in performing binary addition and complementing signal look-ahead, one should evaluate the formation of H_i and H_{i+1} as a function of neighboring bit pairs (i, $i+1$). Let us consider adding two binary numbers A and B together, where

$$A = a_0 2^n + a_1 2^{n-1} + a_2 2^{n-2} + \cdots + a_i 2^{n-i} + \cdots + a_n 2^0 ;$$

$$B = b_0 2^n + b_1 2^{n-1} + b_2 2^{n-2} + \cdots + b_i 2^{n-i} + \cdots + b_n 2^0 .$$

The relation among the new carry (H_i, H_{i+1}) and the neighboring bit pairs (a_i, b_i; a_{i+1}, b_{i+1}) can be expressed as in Table 1 [1]; all of these are generated by a_i, b_i or transmitted through the low-order bits, $i+1$, $i+2$, $\cdots$, with the transmitting-enable switch ON. This signal or new carry can only be terminated when the inhibitor is ON ($a_{i+1} + b_{i+1} = 0$). H_i plays both regular carry and complementing signal roles in performing binary addition.

By grouping all the H_i, we obtain

$$\begin{aligned} H_i &= f(1, 2, 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15) \\ &= a_i b_i + H_{i+1}(\bar{a}_{i+1} b_{i+1} + a_{i+1} \bar{b}_{i+1} + a_{i+1} b_{i+1}) \\ &= a_i b_i + H_{i+1}(a_{i+1} + b_{i+1}) = k_i + H_{i+1} T_{i+1} , \end{aligned} \qquad (1)$$

where k_i is the new complementing signal, H_{i+1} is the previous complementary signal, and T_{i+1} is the previous carry enable switch or the previous stage propagate.

Equation (1) shows that new carry H_i can be formed locally by k_i or produced remotely; H_{i+1} can be produced with the remote stage carry inhibitor not ON ($a_{i+1} + b_{i+1}$

Reprinted with permission from *IBM Journal of Research Development*, H. Ling, "High-Speed Binary Adder," Vol. 25, No. 3, p 156-166, May 1981. © 1981 International Business Machines.

= 1). The formation of sum S_i can be expressed by a similar process. The truth table for S_i is shown in Table 2.

By grouping all the S_i, we obtain

$$S_i = f(1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 14, 15) ;$$

$$1, 2, 3 \rightarrow \bar{a}_i\bar{b}_iH_{i+1}(\bar{a}_{i+1}b_{i+1} + a_{i+1}\bar{b}_{i+1} + a_{i+1}b_{i+1})$$
$$= \bar{a}_i\bar{b}_iH_{i+1}(a_{i+1} + b_{i+1})$$
$$= [a_ib_i + H_{i+1}(a_{i+1} + b_{i+1})]\bar{a}_i\bar{b}_i$$
$$= H_i\overline{T}_i ;$$

$$5, 6, 9, 10 \rightarrow (a_i \forall b_i)(a_{i+1} \forall b_{i+1})\overline{H}_{i+1}$$
$$= (a_i \forall b_i)(a_{i+1} \forall b_{i+1})\overline{H}_{i+1} ;$$

$$4, 8 \rightarrow (a_i \forall b_i)(\bar{a}_{i+1}\bar{b}_{i+1}) ;$$

$$4, 5, 6, 8, 9, 10 \rightarrow (a_i \forall b_i)[\overline{H}_{i+1}(a_{i+1} \forall b_{i+1}) + \bar{a}_{i+1}\bar{b}_{i+1}]$$
$$= (a_i + b_i)(\bar{a}_i + \bar{b}_i)[\overline{H}_{i+1}(a_{i+1} \forall b_{i+1})$$
$$+ \bar{a}_{i+1}\bar{b}_{i+1}\overline{H}_{i+1} + \bar{a}_{i+1}\bar{b}_{i+1}]$$
$$= (a_i + b_i)(\bar{a}_i + \bar{b}_i)(\overline{H}_{i+1} + \bar{a}_{i+1}\bar{b}_{i+1}) ;$$

$$H_i = a_ib_i + H_{i+1}(a_{i+1} + b_{i+1}) ;$$

$$\overline{H}_i = (\bar{a}_i + \bar{b}_i)(\overline{H}_{i+1} + \bar{a}_{i+1}\bar{b}_{i+1}) ;$$

$$4, 5, 6, 8, 9, 10 \rightarrow (a_i + b_i)\overline{H}_i = T_i\overline{H}_i ;$$

$$13, 14, 15 \rightarrow a_ib_iH_{i+1}(\bar{a}_{i+1}b_{i+1} + a_{i+1}\bar{b}_{i+1} + a_{i+1}b_{i+1})$$
$$= a_ib_iH_{i+1}(a_{i+1} + b_{i+1})$$
$$= k_iH_{i+1}T_{i+1} ;$$

$$S_i = f(1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 14, 15)$$
$$= H_i\overline{T}_i + T_i\overline{H}_i + k_iH_{i+1}T_{i+1}$$
$$= (H_i \forall T_i) + k_iH_{i+1}T_{i+1} . \quad (2)$$

We have obtained a set of recursive formulae for both new carry H_i and sum S_i. They are different from the conventional process. Before discovering the difference, let us examine the carry-look-ahead process.

New carry-look-ahead

For ease of discussion, let us consider $i = 31$. We have

$$H_{31} = k_{31} + H_{32}T_{32} . \quad (3a)$$

By substituting $i = 30$, 29, and 28 in (3a), we obtain

$$H_{28} = k_{28} + T_{29}k_{29} + T_{29}T_{30}k_{30} + T_{29}T_{30}T_{31}k_{31}$$
$$+ T_{29}T_{30}T_{31}T_{32}k_{32} . \quad (3b)$$

By following a similar process, we obtain

$$H_{24} = k_{24} + T_{25}k_{25} + T_{25}T_{26}k_{26} + T_{25}T_{26}T_{27}k_{27}$$
$$+ T_{25}T_{26}T_{27}T_{28}H_{28} ;$$

Table 1 The relation of new carry H_i with H_{i+1} and its neighboring bit pairs $(a_i, b_i; a_{i+1}, b_{i+1})$.

i	$H_i = 1$ *in relation with* H_{i+1}	a_i	b_i	a_{i+1}	b_{i+1}
0	$H_i = 0$	0	0	0	0
1	$H_{i+1} = 1$	0	0	0	1
2	$H_{i+1} = 1$	0	0	1	0
3	$H_{i+1} = 1$	0	0	1	1
4	$H_i = 0$	0	1	0	0
5	$H_{i+1} = 1$	0	1	0	1
6	$H_{i+1} = 1$	0	1	1	0
7	$H_{i+1} = 1$	0	1	1	1
8	$H_i = 0$	1	0	0	0
9	$H_{i+1} = 1$	1	0	0	1
10	$H_{i+1} = 1$	1	0	1	0
11	$H_{i+1} = 1$	1	0	1	1
12	$H_{i+1} = X$	1	1	0	0
13	$H_{i+1} = X$	1	1	0	1
14	$H_{i+1} = X$	1	1	1	0
15	$H_{i+1} = X$	1	1	1	1

Table 2 Sum S_i formation.

i	S_i	a_i	b_i	a_{i+1}	b_{i+1}
0	0	0	0	0	0
1	$H_{i+1} = 1$	0	0	0	1
2	$H_{i+1} = 1$	0	0	1	0
3	$H_{i+1} = 1$	0	0	1	1
4	1	0	1	0	0
5	$H_{i+1} = 0$	0	1	0	1
6	$H_{i+1} = 0$	0	1	1	0
7	0	0	1	1	1
8	1	1	0	0	0
9	$H_{i+1} = 0$	1	0	0	1
10	$H_{i+1} = 0$	1	0	1	0
11	0	1	0	1	1
12	0	1	1	0	0
13	$H_{i+1} = 1$	1	1	0	1
14	$H_{i+1} = 1$	1	1	1	0
15	$H_{i+1} = 1$	1	1	1	1

$$H_{20} = k_{20} + T_{21}k_{21} + T_{21}T_{22}k_{22} + T_{21}T_{22}T_{23}k_{23} + T_{21}T_{22}T_{23}T_{24}H_{24}\ ; \quad (4)$$

$$H_{16} = k_{16} + T_{17}k_{17} + T_{17}T_{18}k_{18} + T_{17}T_{18}T_{19}k_{19} + T_{17}T_{18}T_{19}T_{20}H_{20}. \quad (5)$$

By substituting (3b) for (5), we obtain

$$H_{16} = H^*_{16} + I^*_{16}H_{20}, \quad (6)$$

where

$$H^*_{16} = k_{16} + T_{17}k_{17} + T_{17}T_{18}k_{18} + T_{17}T_{18}T_{19}k_{19}\ ; \quad (7)$$

$$I^*_{16} = T_{17}T_{18}T_{19}T_{20}\ . \quad (8)$$

By substituting (3a) and (3b) for (5), we obtain

$$H_{16} = H^*_{16} + I^*_{16}H^*_{20} + I^*_{16}I^*_{20}H^*_{24} + I^*_{16}I^*_{20}I^*_{24}H_{28}\ . \quad (9)$$

The asterisk of H^*_{16} represents the fact that H^*_{16} can be implemented with one level of logic. Based on current switching technology, both fan-in and fan-out are equal to four with eight-emitter dotting; H_{16} can be implemented with two levels of logic.

Comparison with the existing scheme

Based on the local bit pair (a_i, b_i), carry C_i and sum S_i can be written in the form

$$C_i = g_i + C_{i+1}p_i, \qquad g_i = a_ib_i\ ; \quad (10)$$

$$S_i = a_i \forall b_i \forall C_{i+1}, \quad p_i = a_i + b_i\ .$$

For $i = 16$, we have

$$C_{16} = g_{16} + C_{17}p_{16}\ .$$

By substituting $i = 17, 18, \cdots, 19$, C_{16} can be rewritten as

$$\begin{aligned} C_{16} &= g_{16} + p_{16}(g_{17} + p_{17}g_{18} + p_{17}p_{18}g_{19} + p_{17}p_{18}p_{19}C_{20}) \\ &= g_{16} + p_{16}g_{17} + p_{16}p_{17}g_{18} + p_{16}p_{17}p_{18}g_{19} \\ &\quad + p_{16}p_{17}p_{18}p_{19}C_{20} \\ &= G_{16p} + P_{16p}C_{20}\ , \end{aligned} \quad (11)$$

where G_{16p} and P_{16p} are the grouping of the following terms:

$$G_{16p} = g_{16} + p_{16}g_{17} + p_{16}p_{17}g_{18} + p_{16}p_{17}p_{18}g_{19}\ ; \quad (12)$$

$$P_{16p} = p_{16}p_{17}p_{18}p_{19}\ . \quad (13)$$

Similarly, C_{16} can be written in terms of C_{28}:

$$C_{16} = G_{16p} + P_{16p}G_{20p} + P_{16p}P_{20p}G_{24p} + P_{16p}P_{20p}P_{24p}G_{28p}\ .$$

Equations (6), (7), and (8) are similar to (11), (12), and (13); however, H^*_{16} can be implemented with one level of logic, whereas G_{16p} cannot. By expanding (7) and (12) we obtain

$$\begin{aligned} H^*_{16} &= a_{16}b_{16} + (a_{17} + b_{17})a_{17}\,b_{17} \\ &\quad + (a_{17} + b_{17})(a_{18} + b_{18})a_{18}b_{18} \\ &\quad + (a_{17} + b_{17})(a_{18} + b_{18})(a_{19} + b_{19})a_{19}b_{19} \\ &= a_{16}b_{16} + a_{17}b_{17} + a_{17}a_{18}b_{18} + b_{17}a_{18}b_{18} \\ &\quad + a_{17}a_{18}a_{19}b_{19} + a_{17}b_{18}a_{19}b_{19} \\ &\quad + b_{17}a_{18}a_{19}b_{19} + b_{17}b_{18}a_{19}b_{19}\ ; \end{aligned} \quad (14)$$

$$\begin{aligned} G_{16p} &= a_{16}b_{16} + (a_{16} + b_{16})a_{17}b_{17} \\ &\quad + (a_{16} + b_{16})(a_{17} + b_{17})a_{18}b_{18} \\ &\quad + (a_{16} + b_{16})(a_{17} + b_{17})(a_{18} + b_{18})a_{19}b_{19} \\ &= a_{16}b_{16} + a_{16}a_{17}b_{17} + b_{16}a_{17}b_{17} \\ &\quad + a_{16}a_{17}a_{18}b_{18} + a_{16}b_{17}a_{18}b_{18} \\ &\quad + b_{16}a_{17}a_{18}b_{18} + b_{16}b_{17}a_{18}b_{18} + a_{16}a_{17}a_{18}a_{19}b_{19} \\ &\quad + a_{16}a_{17}b_{18}a_{19}b_{19} + a_{16}b_{17}a_{18}a_{19}b_{19} + a_{16}b_{17}b_{18}a_{19}b_{19} \\ &\quad + b_{16}a_{17}a_{18}a_{19}b_{19} + b_{16}a_{17}b_{18}a_{19}b_{19} \\ &\quad + b_{16}b_{17}a_{18}a_{19}b_{19} + b_{16}b_{17}b_{18}a_{19}b_{19}\ . \end{aligned} \quad (15)$$

Equation (14) contains eight terms, whereas (15) contains fifteen. With current available technology, the former can be implemented with one level of logic (this is shown in detail in the next section); the latter can only be implemented with two levels of logic.

Let us further examine the ith-digit carry formation. For (1), the carry is generated by local complementing signal k_i, and the remote carry H_{i+1} is controlled by remote bit pair $(a_{i+1} + b_{i+1})$; whereas for (10), the carry is generated by local carry g_i, and the remote carry C_{i+1} is controlled by local bit pair $(a_i + b_i)$. From the carry-look-ahead point of view, (1) offers faster resolution, whereas the latter is one stage slower. That is why (14) contains only eight terms, and (15), fifteen.

To illustrate the step-by-step operation, two examples are given.

Example 1 Assume the contents of A and B registers to be as shown and find their sum;

A register 00000000011010101111101100011001

B register 00000000011011011101010101010111

The k_i and T_i can be implemented with one level of logic:

k_i 00000000011010001101000100010001

T_i 00000000011011111111111101011111

The complementary signals can be implemented by grouping k_i and T_i together. This process requires one level of logic:

H_i 00000000111111111111111100111111

The sum digit S_i is implemented in parallel with H_i; the result of H_i will force S_i to select one value between H_i = 0 and H_i = 1:

S_i 00000000110110001101000001110000

This example demonstrates that it is possible to implement a 32-bit adder with three levels of logic with the hardware constraints indicated in the previous section. The detailed implementation of S_i is discussed in the next section.

Example 2 Assuming that the contents of index, base, and displacement registers are as shown, compute the virtual address. (To test the generality of this scheme, odd contents are purposely chosen; in the normal mode of operation, an EXCPN will occur.)

Index register 00000000000111010111011101011011

Base register 00000000000001011100100111011101

Displacement register 101111010111

To implement the carry-save adder (CSA) requires one level of logic:

s_i 00000000000110001111010101010001

c_i 00000000000001010000101111010111110

Implementation of k_i and T_i requires one additional level of logic:

k_i 00000000000010000001010100010000

T_i 00000000000110101111011111111111

Implementation of the complementary signal requires one level of logic to group k_i and T_i together:

H_i 00000000001110011111111111110000

The address digit S_i is implemented in parallel with H_i; the result of H_i will force S_i to select one value between H_i = 0 and 1:

S_i 00000000001000110000110100001111

This example demonstrates that the AGEN adder can be implemented with four rather than six levels of logic, as is the case in current machine organization. The detailed implementation of S_i (the address) is discussed in the next section. The logic implementation of every fourth bit (i = 31, 27, 23, 19, 16, 15, 11, 7, 3, 0) is shown in the Appendix of this paper.

Implementation

The detailed implementation can be divided into two categories: binary addition and subtraction, and address generation.

- *Addition and subtraction*

Equation (3b) is a general representation of the new carry-look-ahead process. For ADDITION, $k_{32} = 0$; therefore, the fifth term in (3b) is dropped and H_{28} can be written as

$$H_{28} = k_{28} + T_{29}k_{29} + T_{29}T_{30}k_{30} + T_{29}T_{30}T_{31}k_{31} . \quad (4a)$$

For SUBTRACTION, there is a HOT ONE carry input from bit 31; thus H_{28} can be written as

$$H_{28} = k_{28} + T_{29}k_{29} + T_{29}T_{30}k_{30} + T_{29}T_{30}T_{31}k_{31} + T_{29}T_{30}T_{31} . \quad (4b)$$

Equation (2) shows that S_i is a function of H_i and H_{i+1}. For ease of implementation, this equation is rewritten in the form

$$\begin{aligned} S_i &= (H_i \forall T_i) + k_i H_{i+1} T_{i+1} \\ &= [(k_i + H_{i+1}T_{i+1}) \forall T_i] + k_i H_{i+1} T_{i+1} \\ &= H_{i+1}(\bar{T}_i T_{i+1} + k_i T_{i+1}) \\ &\quad + \bar{H}_{i+1}\bar{k}_i T_i + k_i \bar{T}_i + \bar{k}_i T_i \bar{T}_{i+1} . \end{aligned} \quad (16)$$

Equation (16) demonstrates that S_i can be written in the conditional form

$$S_i(H_{i+1} = 0) = k_i \forall T_i ;$$

$$S_i(H_{i+1} = 1) = \bar{T}_i T_{i+1} + k_i T_{i+1} + k_i \bar{T}_i + \bar{k}_i T_i \bar{T}_{i+1} .$$

The general expression of SUM S_i can be written as

$$S_i = H_{i+1}(a_{i+1} + b_{i+1})\overline{(a_i \forall b_i)} + \bar{H}_{i+1}(a_i \forall b_i) + \overline{(a_{i+1} + b_{i+1})}(a_i \forall b_i) .$$

For i = 31, we have

$$S_{31} = H_{32}(a_{32} + b_{32})\overline{(a_{31} \forall b_{31})} + \bar{H}_{32}(a_{31} \forall b_{31}) + \overline{(a_{32} + b_{32})}(a_{31} \forall b_{31}) .$$

For i = 0, we obtain

$$S_0 = H_1(a_1 + b_1)\overline{(a_0 \forall b_0)} + \bar{H}_1(a_0 \forall b_0) + \overline{(a_1 + b_1)}(a_0 \forall b_0) ; \quad (17)$$

$$\begin{aligned} H_1 &= k_1 + T_2 k_2 + T_2 T_3 k_3 + T_2 T_3 T_4 H_4 \\ &= H_1^* + I_1^* H_4 ; \end{aligned} \quad (18)$$

$$H_4 = k_4 + T_5k_5 + T_5T_6k_6 + T_5T_6T_7k_7 + T_5T_6T_7T_8H_8$$
$$= H^*_4 + I^*_4H_8 ; \quad (19)$$

$$H_8 = H^*_8 + I^*_8H_{12} ; \quad (20)$$

$$H_{12} = H^*_{12} + I^*_{12}H_{16} . \quad (21)$$

By substituting (21) for (20), we have

$$H_8 = H^*_8 + I^*_8H^*_{12} + I^*_8I^*_{12}H_{16} . \quad (22)$$

Similarly, we obtain H_4 and H_1:

$$H_4 = H^*_4 + I^*_4H^*_8 + I^*_4I^*_8H^*_{12} + I^*_4I^*_8I^*_{12}H_{16}; \quad (23)$$

$$H_1 = H^*_1 + I^*_1H^*_4 + I^*_1I^*_4H^*_8 + I^*_1I^*_4I^*_8H^*_{12} + I^*_1I^*_4I^*_8I^*_{12}H_{16}. \quad (24)$$

By substituting Eqs. (22), (23), and (24) for Eqs. (18), (19), (20), and (21), Eq. (17) can be written as

$$\begin{aligned} S_0 &= (H^*_1 + I^*_1H^*_4 + I^*_1I^*_4H^*_8 + I^*_1I^*_4I^*_8H^*_{12} \\ &\quad + I^*_1I^*_4I^*_8I^*_{12}H_{16})(a_1 + b_1)\overline{(a_0 \forall b_0)} \\ &\quad + \overline{(H^*_1 + I^*_1H^*_4 + I^*_1I^*_4H^*_8 + I^*_1I^*_4I^*_8H^*_{12}} \\ &\quad + \overline{I^*_1I^*_4I^*_8I^*_{12}H_{16})}(a_0 \forall b_0) + \overline{(a_1 + b_1)}(a_0 \forall b_0) ; \\ &= (H^*_1 + I^*_1H^*_4 + I^*_1I^*_4H^*_8 + I^*_1I^*_4I^*_8H^*_{12})(a_1 + b_1) \\ &\quad \times \overline{(a_0 \forall b_0)} + I^*_1I^*_4I^*_8I^*_{12}H_{16}(a_1 + b_1)\overline{(a_0 \forall b_0)} \\ &\quad + \overline{(H^*_1 + I^*_1H^*_4 + I^*_1I^*_4H^*_8 + I^*_1I^*_4I^*_8H^*_{12})} \\ &\quad \times (\overline{I^*_1I^*_4I^*_8I^*_{12}} + \bar{H}_{16})(a_0 \forall b_0) \\ &\quad + \overline{(a_1 + b_1)}(a_0 \forall b_0) . \end{aligned} \quad (25)$$

By using the Sklansky conditional-sum method [2], (25) can be written as

$$\begin{aligned} S_0 &= (H^*_1 + I^*_1H^*_4 + I^*_1I^*_4H^*_8 + I^*_1I^*_4I^*_8H^*_{12}) \\ &\quad \times (a_1 + b_1)\overline{(a_0 \forall b_0)} + \overline{(a_1 + b_1)}(a_0 \forall b_0) \\ &\quad + \overline{(H^*_1 + I^*_1H^*_4 + I^*_1I^*_4H^*_8 + I^*_1I^*_4I^*_8H^*_{12})} \\ &\quad \times (a_0 \forall b_0) \quad [H_{16} = 0] \\ &\quad + I^*_1I^*_4I^*_8I^*_{12}(a_1 + b_1)\overline{(a_0 \forall b_0)} \quad [H_{16} = 1] \\ &\quad + \overline{(H^*_1 + I^*_1H^*_4 + I^*_1I^*_4H^*_8 + I^*_1I^*_4I^*_8H^*_{12})} \\ &\quad \times \overline{(I^*_1I^*_4I^*_8I^*_{12})}(a_0 \forall b_0) \quad [H_{16} = 1]. \end{aligned}$$

The hardware implementation of S_0 is included in the Appendix.

Equation (9) has indicated that H_{16} can be implemented with two levels of logic. Let us examine the individual terms of S_0. It is clearly pointed out that they also require only two levels of logic to be implemented. That is to say, when H_{16} is ready, S_0 can be obtained by using one additional level of logic. We have proved, by using current switching logic, that one can implement a 32-bit adder by consuming only three levels of logic.

- *Address generation*

In the address generation process, we are dealing with positive numbers only. Therefore, $k_{32} = 0$. The output of the (3, 2) carry-save adder provides the s_i and c_{i+1} corresponding to the a_i and b_i bit pairs. In addition, X_i and B_i both are 32 bits in length. However, D_i has only a 12-bit width. For i = 0–18, the output of the carry-save adder has a special pattern; s_i and c_{i+1} will not have the form

111—101—1111—11

101—111—1111—11 .

In general, the output of CSA will appear as

1111—01—001—10110

0001—01—001—10010 .

Therefore, for i = 19–31, S_i appears as usual:

$$S_i = (H_i \forall T_i) + k_iH_{i+1}T_{i+1} .$$

For i = 0–18, S_i appears as

$$S_i = (H_i \forall T_i) .$$

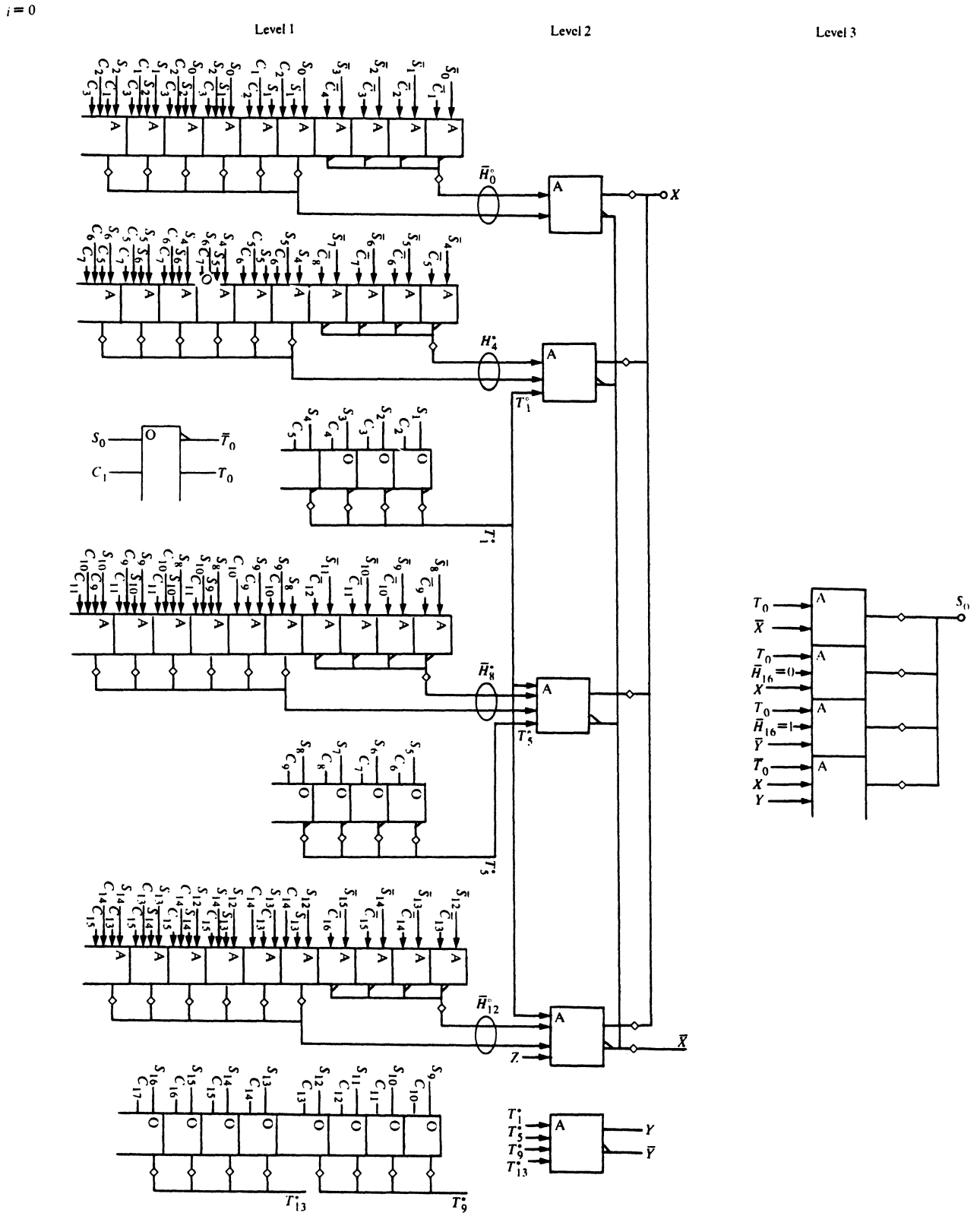
The detailed implementations for i from 0, 3, 7, 11, 15, 16, 19, 23, 27, and 31 are shown in the Appendix.

Summary

It is intended in this paper to speed up the carry propagation for examining two bit pairs. The formulation of H^*_{16} contains eight terms as compared to that of the regular carry-look-ahead process, where G_{16p} contains fifteen terms. It is possible to implement H^*_{16} with one level of logic, whereas it is not possible with G_{16p}. The formulation of sum S_i in this new process will contain slightly more terms; however, they are not in the critical path.

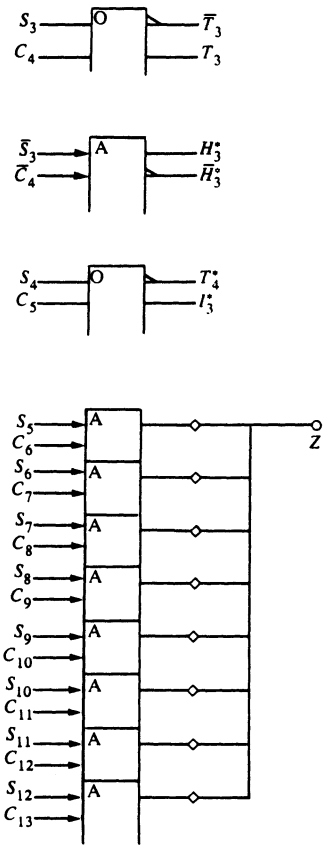
References

1. H. Ling, "High Speed Binary Parallel Adder," *IEEE Trans. Electron. Computers* **EC-15**, 799-802 (1966).
2. J. Sklansky, "Conditional-Sum Addition Logic," *IEEE Trans. Electron. Computers* **EC-9**, 226-231 (1960).

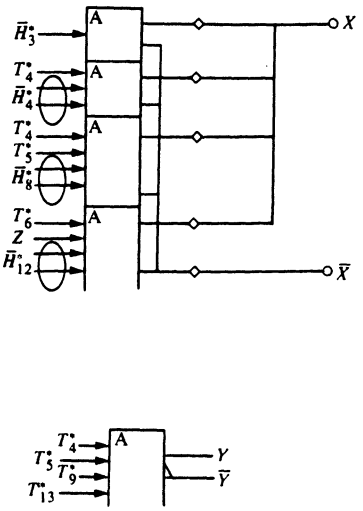

i = 0
Level 1
Level 2
Level 3
A
O
X
Y
Z

$i = 3$

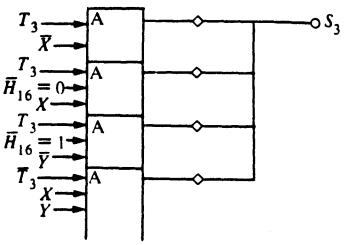
Level 1



Level 2

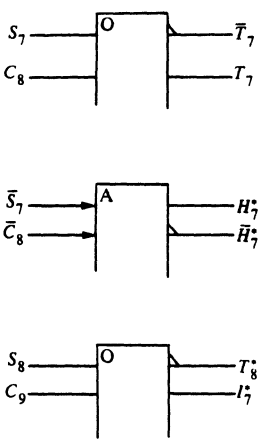


Level 3

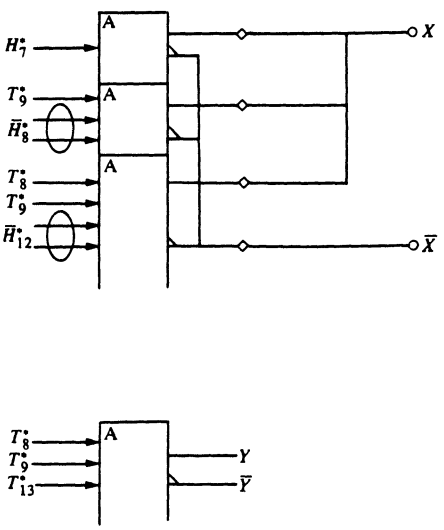


$i = 7$

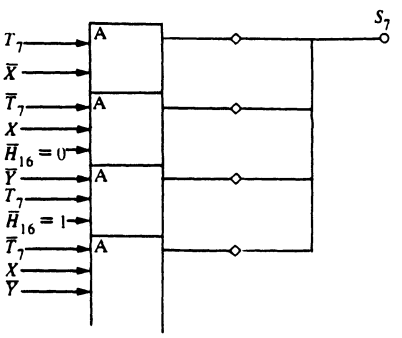
Level 1



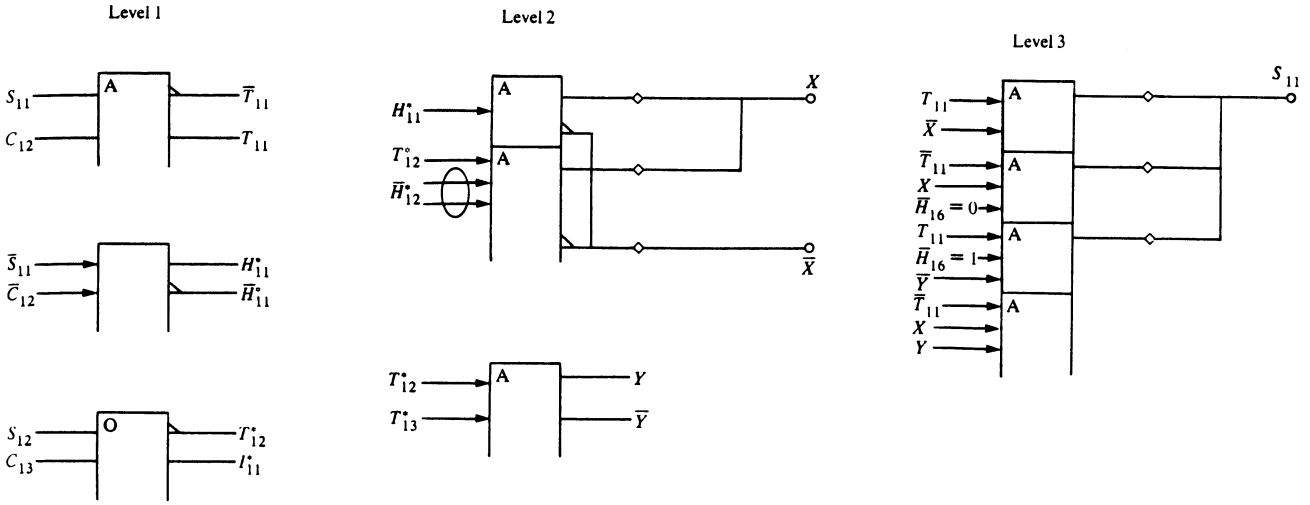
Level 2



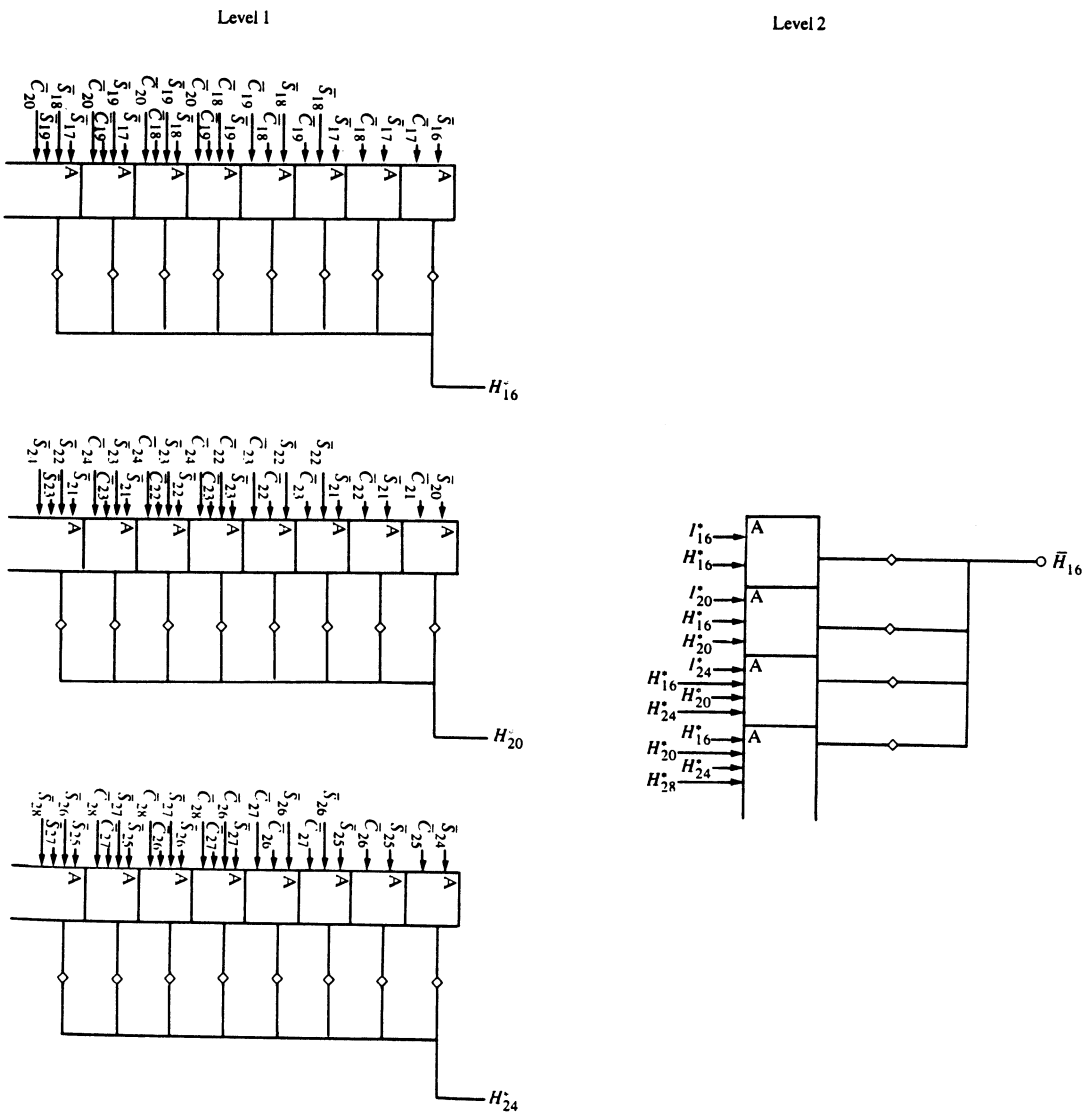
Level 3



$i = 11$

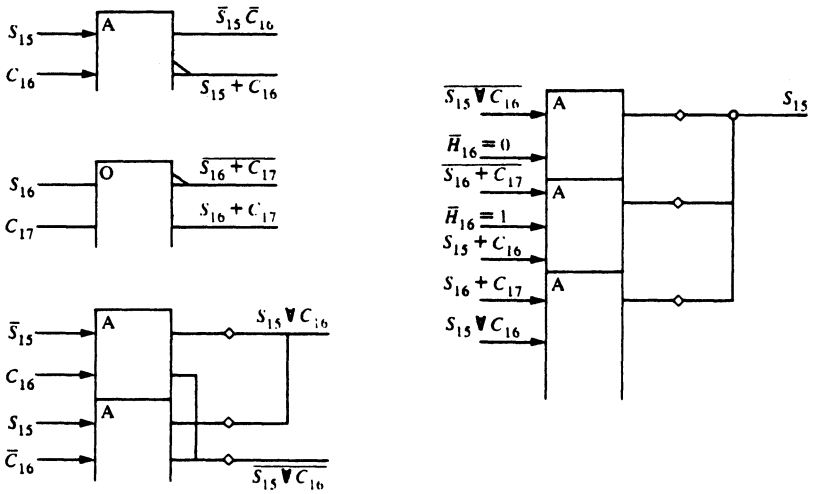

Level 1
Level 2
Level 3
A
O
S_{11}
C_{12}
$\overline{T}_{11}$
T_{11}
$\overline{S}_{11}$
$\overline{C}_{12}$
H^*_{11}
$\overline{H}^*_{11}$
S_{12}
C_{13}
T^*_{12}
I^*_{11}
H^*_{11}
T^*_{12}
$\overline{H}^*_{12}$
X
$\overline{X}$
T^*_{12}
T^*_{13}
Y
$\overline{Y}$
T_{11}
$\overline{X}$
$\overline{T}_{11}$
X
$\overline{H}_{16} = 0$
T_{11}
$\overline{H}_{16} = 1$
$\overline{Y}$
$\overline{T}_{11}$
X
Y
S_{11}

$i = 16$


Level 1
Level 2
Level 3
A
H^*_{16}
H^*_{20}
H^*_{24}
I^*_{16}
H^*_{16}
I^*_{20}
H^*_{16}
H^*_{20}
I^*_{24}
H^*_{16}
H^*_{20}
H^*_{24}
H^*_{16}
H^*_{20}
H^*_{24}
$\overline{H}_{16}$

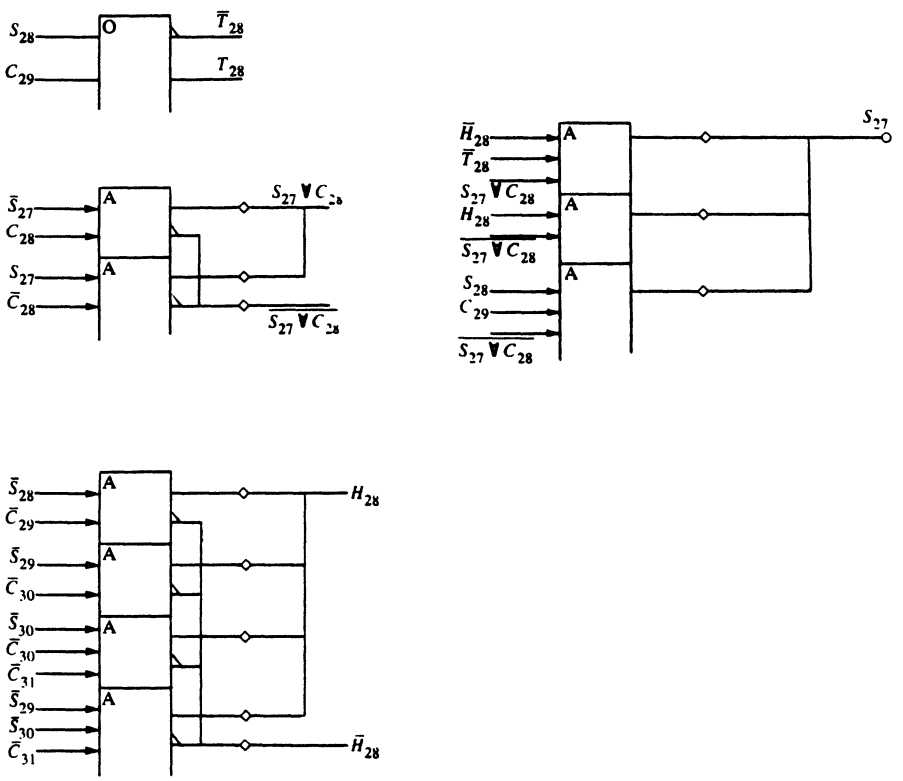
$i = 15$

Level 1 Level 2 Level 3

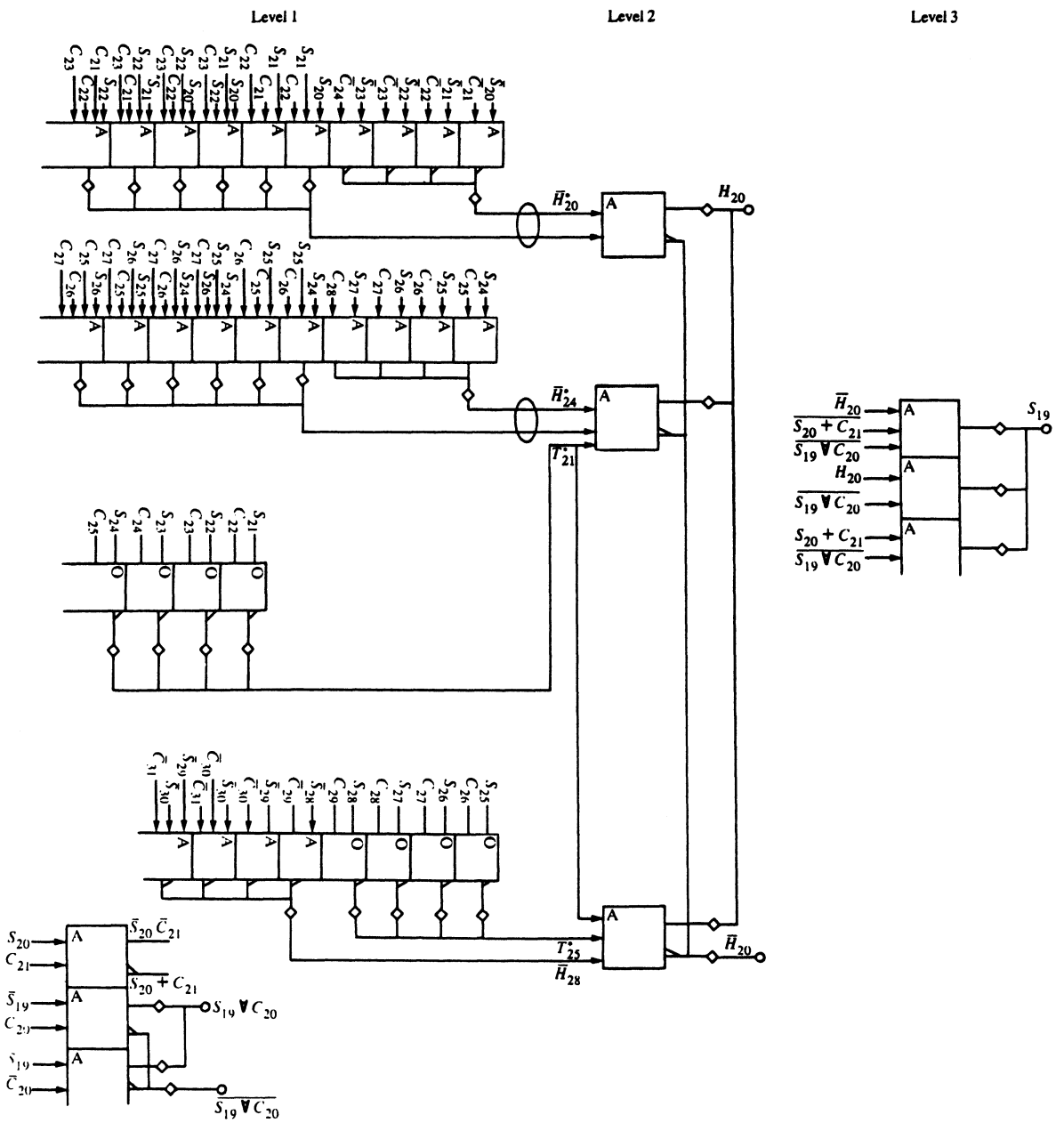


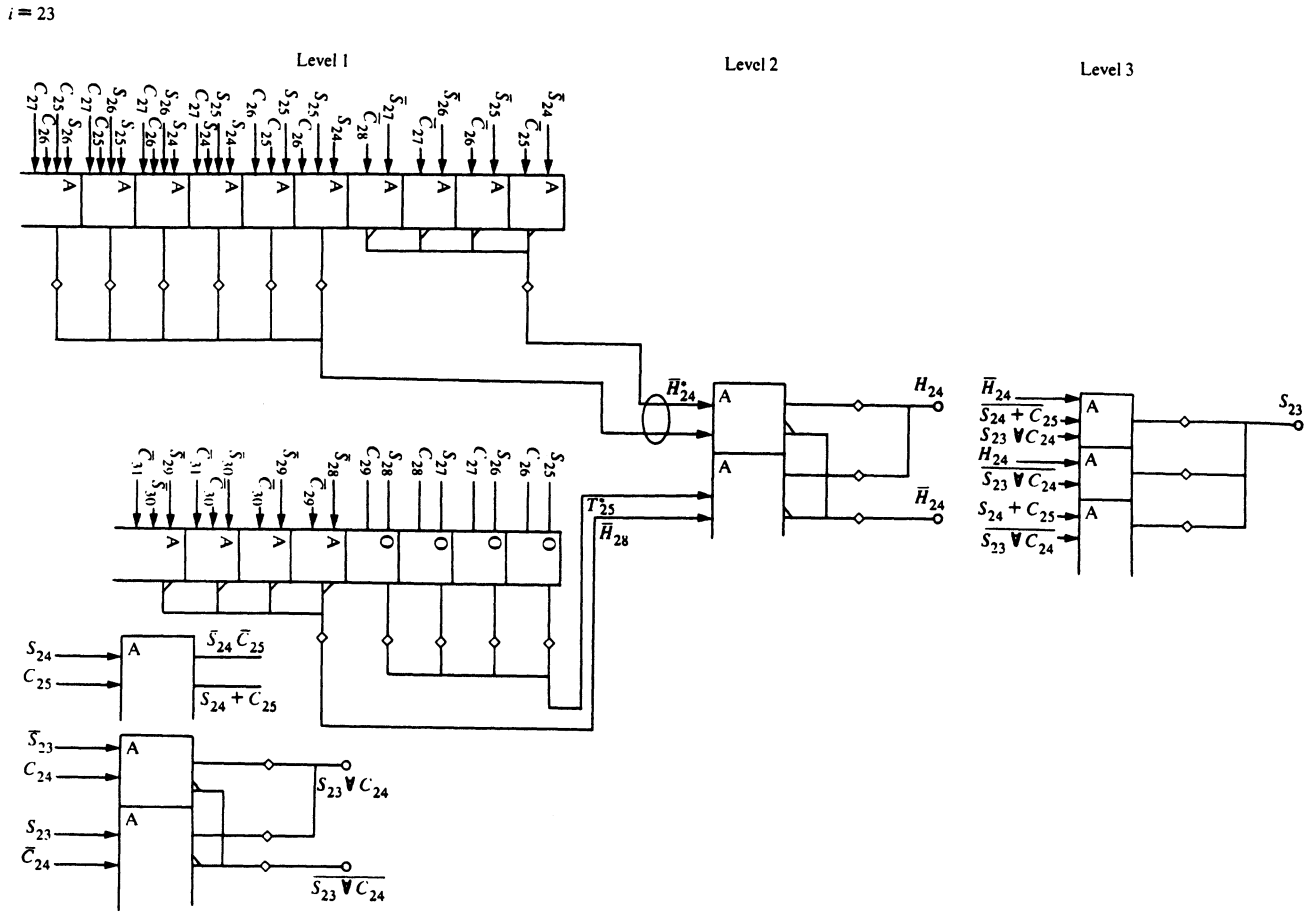
$i = 27$

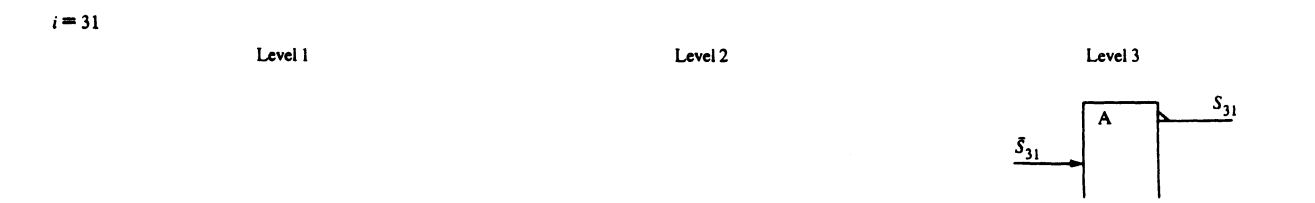
Level 1 Level 2 Level 3



$i = 19$


Level 1
Level 2
Level 3
A
O
$\bar{H}^*_{20}$
H_{20}
$\bar{H}^*_{24}$
T^*_{21}
T^*_{25}
$\bar{H}_{28}$
$\bar{H}_{20}$
S_{19}
$S_{20} + C_{21}$
$S_{19} \forall C_{20}$
$\bar{S}_{20}\bar{C}_{21}$
$\bar{S}_{19}\forall C_{20}$





Received September 11, 1980; revised November 26, 1980

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

A Sub-Nanosecond 0.5µm 64b Adder Design

SAMUEL NAFFZIGER

HEWLETT-PACKARD CO., FORT COLLINS, CO

A sub-nanosecond 64b adder in 0.5µm CMOS forms the basis for the integer and floating point execution units. Integrating dual-rail dynamic CMOS and use of Ling's equations, the adder is composed of 7k FETs in 0.246mm² and performs a full 64b add, operands to result in <1ns (7 fanout of 4 inverter delays) under nominal conditions.

Addition time is important to CPU design. Add latency is in the critical path of such important areas as memory address calculation, ALU evaluation and floating point computation. Current state of the art addition hardware [1-3] is not adequate for 64b computing, and this prompted a new design that forms the core of the execution hardware. It incorporates architecture and circuit design techniques that enable it to achieve low latency for adders of width from 13b to 112b. These techniques for the 64b adder design are described here.

Conceptually, an adder is:

$$Sum = A \oplus B \oplus Carry\_in$$
$$Carry = (A * B) + (A + B) * Carry\_in$$

Rippling the carry one bit at a time however is clearly unacceptable for a 64b adder so a fusion of carry-look ahead and carry-select techniques is implemented. Carry-look-ahead techniques involve parallel calculation of groups of carries (in this case, 4 at a time) in a modular fashion that reduces the carry calculation time to $Log_r[n] + 2$ gate delays where r is the group size and n is the width of the adder. Most current adder designs are based on this scheme [1, 2]. The equations for a group of 4 carry look-ahead are:

$$C4 = G3 + G2*P3 + G1*P2*P3 + G0*P1*P2*P3$$

Where the G and P terms are the generate and propagate terms derived from the operands (G=A*B, P=A+B). These groups of carries can then be combined at another level to produce a longer carry look-ahead using the same equations. Hence, for a traditional group of 2 carry look-ahead adder delay:
1 (generate GPK terms) + $\log_2[64]$ + 1 (Final XOR) = 8 gate delays.
If the technology of implementation allows the greater fanout and fanin of group of 4 carry look-ahead, the gate delays are:

$$1 + \log_4[64] + 1 = 5 \text{ gate delays.}$$

A sum select method is often used that goes ahead and calculates two sums based on Carry_in = 0 and 1 for a group using duplicate carry chains. When the actual carry into that group becomes available via look ahead, the correct sum is selected.

Ling's equations can be used to reduce delay further. Ling developed a series of equations specifically to make use of the dot or capability of ECL logic [4]. The result is the definition of a psuedo-carry calculated quickly with dot-or circuits. The relevant equations are as follows for a normal 4b carry:

$$C4 = G3 + G2*P3 + G1*P2*P3 + G0*P1*P2*P3$$

For a Ling "psuedo carry" [5]

$$H4 = G3 + G2*P2 + G1*P1*P2 + G0*P2*P1*P0$$

The propagate terms are simply shifted by one so C4 = H4*P3. When we define P as the OR of the operands, the G2*P2 term is redundant since G2 implies P2. This redundancy can be used to simplify all of the terms for H4 yielding an equation based on the actual operands, not P and G terms and can be calculated in one gate delay with fanin 4 logic:

$$H4 = A3*B3 + A2*B2 + A2*A1*B1 + A1*B2*B1 + A2*A1*A0*B0 + A2*B1*A0*B0 + B2*A1*A0*B0 + B2*B1*A0*B0$$

This is simpler (8 terms, fanin of 4 vs.15 terms, fanin of 5) than the expansion of the C4 equation in terms of the operands. With careful use of X terms in the carry propagation path, the conversion of H4 into C4 can be accomplished with no penalty.

To perform a CLA, 4b propagate terms must be calculated in at least the same time as the H4 terms that can then be combined for higher level look-aheads. These 4b propagate terms (I4 = P3*P2*P1*P0) can be calculated using a wired-or to generate P_n. This wired-or capability can be duplicated in dynamic CMOS using multiple NFET pulldown legs on a precharged node. The resulting circuits that generate H4 and I4 directly off the input operands are shown in Figure 1. The H4 and I4 terms are combined in a distributed Manchester gate to produce the block of 16 carry signals:

$$C16 = (H0+I0)*I1*I2*I3 + H1*I2*I3 + H2*I3 + H3$$

Where the Hx Ix terms are for the x'th group of 4 in the 16b quadrant. These 16b carries are combined in 1 more gate to produce the final carry select signals to the upper quadrants of the 64b adder. In parallel with this long carry generation, a carry ripple is performed in each of the 16b quadrants that generates both values of the carry to be selected. To hide the psuedo-carry to real-carry conversion, the carry chain is shifted over one bit and a special C3 (or carry out of 3b) term is used from the H4 gate in combination with G0 (Figure 2). In this way the I4 terms can be used directly in the carry-ripple chain as well. Finally, the carry select and sum generate are performed in a single gate (Figure 3). The critical path in the 64b adder (Figure 4) involves 4 total gate delays: (H4/I4 generation) + (C16 gen) + (Long carry gen) + (Sum select). The fast fanout of 1 carry ripple that occurrs in parallel with the look-ahead, produces the local carries just ahead of the long carry select signal at each sum gate.

The high gain and fanin capability of dynamic CMOS along with the flexibility of dual rail are key enablers to the design. The adder requires dual monotonic (DCVS) inputs and produces a dual monotonic sum. The number of transistors for a 64b sum is 6924, that when layed out in 0.6µm geometry occupies 96x2560µm or 0.246mm² in 3 layers of metal. This adder occupies several critical paths in a microprocessor whose frequency of operation confirms simulated time at nominal process and voltage conditions of 0.93ns data in to sum out (Figure 5) [6].

References:

[1] Inoue, A., et al., "A 0.4mm 1.4ns 32b Dynamic Adder using Non-precharge Multiplexers and Reduced Precharge Voltage Technique," Symp. VLSI Circuits Digest of Tech. Papers, pp. 9-10, June, 1995.

[2] Dobberpuhl, et al., "A 200 MHz 64b dual-issue CMOS microprocessor," IEEE J. of Solid-State Circuits, Vol. 27, No. 11, Nov., 1992.

[3] Suzuki, M., et al., "A 1.4ns 32b CMOS ALU in Double Pass-Transistor Logic," IEEE J. of Solid-State Circuits, Vol. 28, No. 11, Nov., 1993.

[4] Ling, H., "High Speed Binary Adder," IBM J. Reasearch. Dev., Vol. 25, No. 3, p.156, May, 1981.

[5] Flynn, M., "Topics in Arithmetic For Digital Systems Designers," (Preliminary Second Edition) pp. 104-105, 1995.

[6] Heikes, C., G. Colon-Bonet, "Dual floating-Point Coprocessor with an FMAC Architecture," ISSCC Digest of Technical Papers, pp. 354-356, Feb., 1996.

Reprinted from *Digest of Technical Papers, 1996 IEEE International Solid-State Circuits Conference,* pp. 362-363, February 1996.

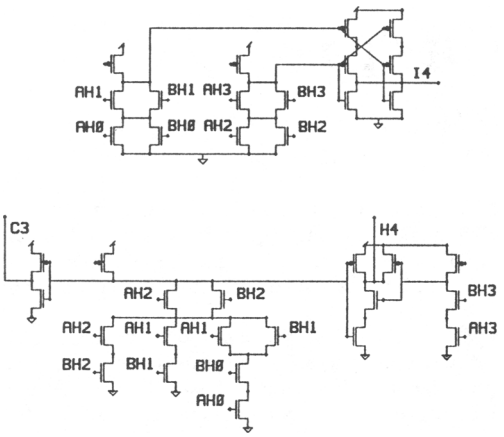


Figure 1: H4 and I4 generation circuits.

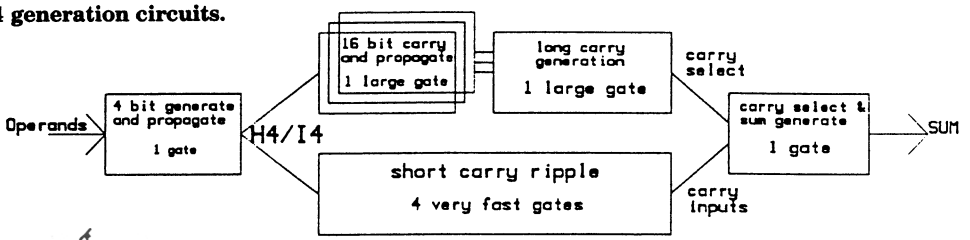


Figure 4: 64b add critical path.

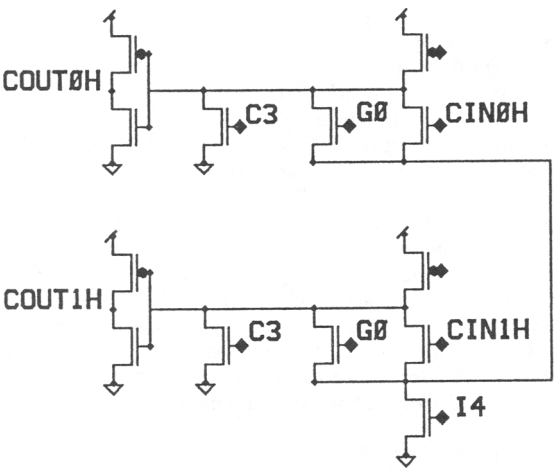


Figure 2: Carry ripple circuit.

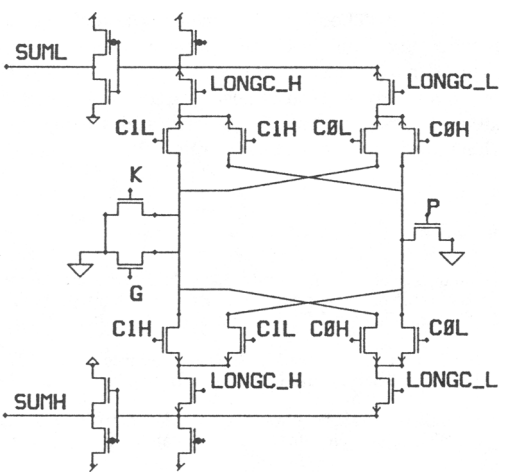


Figure 3: Sum select gate.

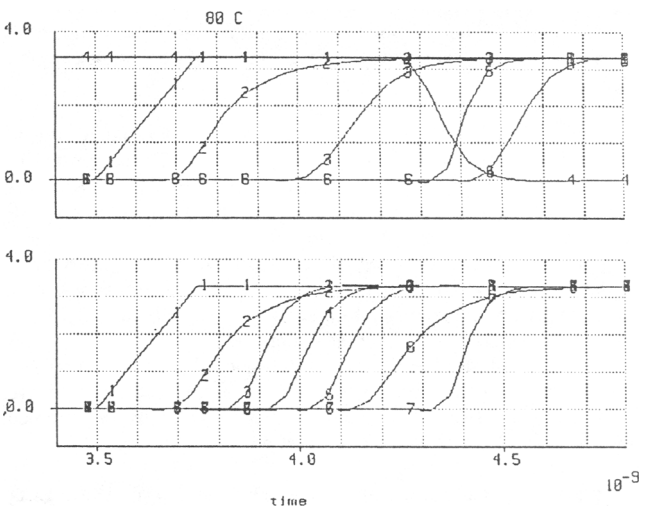


Figure 5: Critical path SPICE waveforms:

Top key: (1) operands, (2) I4 , (3) C16, (5) long carry select, (6) sum.

Bottom key: (1) operands, (2) I4, (6) short carry ripple, (7) long carry select.

A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations

PETER M. KOGGE AND HAROLD S. STONE

Abstract—An mth-order recurrence problem is defined as the computation of the series $x_1, x_2, \cdots, x_N$, where $x_i = f_i(x_{i-1}, \cdots, x_{i-m})$ for some function f_i. This paper uses a technique called recursive doubling in an algorithm for solving a large class of recurrence problems on parallel computers such as the Illiac IV.

Recursive doubling involves the splitting of the computation of a function into two equally complex subfunctions whose evaluation can be performed simultaneously in two separate processors. Successive splitting of each of these subfunctions spreads the computation over more processors.

This algorithm can be applied to any recurrence equation of the form $x_i = f(b_i, g(a_i, x_{i-1}))$ where f and g are functions that satisfy certain distributive and associative-like properties. Although this recurrence is first order, all linear mth-order recurrence equations can be cast into this form. Suitable applications include linear recurrence equations, polynomial evaluation, several nonlinear problems, the determination of the maximum or minimum of N numbers, and the solution of tridiagonal linear equations. The resulting algorithm computes the entire series $x_1, \cdots, x_N$ in time proportional to $\lceil \log_2 N \rceil$ on a computer with N-fold parallelism. On a serial computer, computation time is proportional to N.

Index Terms—Parallel algorithms, parallel computation, recurrence problems, recursive doubling.

Manuscript received October 7, 1972; revised March 21, 1973. This work was supported in part by NSF Grant GJ 1180 and in part by an IBM Corporation fellowship.

P. M. Kogge was with the Department of Electrical Engineering, Digital Systems Laboratory, Stanford University, Stanford, Calif. He is now with the Systems Architecture Department, IBM Corporation, Owego, N.Y. 13827.

H. S. Stone is with the Department of Electrical Engineering and the Department of Computer Science, Digital Systems Laboratory, Stanford University, Stanford, Calif.

INTRODUCTION

A. Definition of Problem

IT FREQUENTLY occurs in applied mathematics that the solution to some problem is a sequence $x_1, x_2, \cdots, x_N$, where each x_i is a function of the previous m x's, namely $x_{i-1}, \cdots, x_{i-m}$. A common example of such a problem is a time-varying linear system, where the state of the system at time i is x_i, and can be computed from the equations

Reprinted from *IEEE Transactions on Computers*, Vol. SC-22, No. 8, pp. 786-793, August 1973.

$$
\begin{aligned}
x_1 &= B_1 \\
x_2 &= A_2 x_1 + B_2 \\
x_3 &= A_3 x_2 + B_3 \\
&\cdot \\
&\cdot \\
&\cdot \\
x_i &= A_i x_{i-1} + B_i \\
&\cdot \\
&\cdot \\
&\cdot \\
x_N &= A_N x_{N-1} + B_N
\end{aligned} \tag{1}
$$

where A_i and B_i represent the internal dynamics of the system. A_i and B_i can be real or complex numbers, constant or time-varying matrices, etc., depending on the problem.

The equation used to compute x_i is called a recurrence equation and, together with some initial values for some of the x_i, represents a complete problem description. Formally, a recurrence problem consists of a set of recurrence equations:

$$x_i = f_i(x_{i-1}, \cdots, x_{i-m}), \qquad i = m+1, \cdots, N \tag{2}$$

and some boundary values, which may consist of the following.

1) $x_1, \cdots, x_m$. This is an initial value problem.
2) $x_{N-m+1}, \cdots, x_N$. This is a final value problem.
3) A mixture of m initial and final values.

This paper discusses an algorithm for solving a particular class of initial value recurrence problems on parallel computing systems such as the Illiac IV. This class of problems includes the computation of the sequence $x_1, \cdots, x_N$ when the expression for x_i is a linear recurrence equation of the form of (1), the calculation of the maximum or minimum of N numbers, the evaluation of Nth-degree polynomials, and several nonlinear problems. Such problems as these can be solved in a very straightforward manner on serial processors in time proportional to N. Some have also been solved on parallel computers with special-purpose algorithms tailored to those problems, e.g., polynomial evaluation (Munro and Paterson [3]). With a computer having N-fold parallelism, the algorithm in this paper solves all these problems and others in time proportional to $\lceil \log_2 N \rceil$.[1]

B. Computer Model

The algorithm to be described in this paper is designed for a computer of the Illiac IV class. The major assumptions about the computer's architecture are as follows.

Assumption 1: There are p identical processors, each able to execute the usual arithmetic and logical operations, and each with its own memory.

Assumption 2: Each processor can communicate with every other processor. The exact method of data exchange between processors can affect the algorithm's computational complexity and will be discussed in a future report.

[1] $\lceil x \rceil$ is the ceiling function and represents the smallest integer not smaller than x.

Assumption 3: Each processor has a distinct index by which it is referenced.

Assumption 4: All processors obtain their instructions simultaneously from a single instruction stream. Thus all processors execute the same instruction, but they operate on data stored in their own memories.

Assumption 5: Any processor may be "blocked" or "masked" from performing some instruction. This mask may be set by an explicit instruction directed to that processor via its index, or by the result of some test instruction such as "set mask if accumulator = 0."

Assumption 6: Elementary arithmetic operations have two operands.

It is assumed throughout this paper that the number of processors p is greater than N, the maximum number of elements to be computed. In reality when p is less than N, this algorithm can be used $\lceil N/p \rceil$ times to calculate p elements of the series at a time.

II. General First-Order Recurrence Equation

A. Example

In this section we develop a parallel solution to a simple first-order recurrence problem. The solution is a special case of the general algorithm, but its development is not obscured by the notation needed to describe the general algorithm.

Given $x_1 = b_1$, find $x_2, \cdots, x_N$, where

$$x_i = a_i x_{i-1} + b_i. \tag{3}$$

Before solving this problem let us give the following definition.

Definition: The function $\hat{Q}(m, n)$, $n \leqslant m$, is defined to be

$$\hat{Q}(m, n) = \sum_{j=n}^{m} \left(\prod_{r=j+1}^{m} a_r \right) b_j$$

where the vacuous product $(\Pi_{r=m+1}^{m} a_r)$ is given the value 1. Stone [4] first used this notation in the derivation of this algorithm. The basic algorithm involves a concept called *recursive doubling*, which consists of breaking the calculation of one term into two equally complex subterms.

Now we can write the solution to (3) as follows:

$$
\begin{aligned}
x_1 &= b_1 && = \hat{Q}(1, 1) \\
x_2 &= a_2 x_1 + b_2 && = a_2 b_1 + b_2 = \hat{Q}(2, 1) \\
x_3 &= a_3 x_2 + b_3 && = a_3 a_2 b_1 + a_3 b_2 + b_3 = \hat{Q}(3, 1) \\
&\cdot \\
&\cdot \\
&\cdot \\
x_i &= a_i x_{i-1} + b_i && = \hat{Q}(i, 1) \\
&\cdot \\
&\cdot \\
&\cdot \\
x_N &= a_N x_{N-1} + b_N = \hat{Q}(N, 1).
\end{aligned} \tag{4}
$$

We can also write this solution as

$$\hat{Q}(1, 1) = x_1 = b_1$$

$$\hat{Q}(2,1) = x_2 = a_2 x_1 + b_2 = a_2 \hat{Q}(1,1) + \hat{Q}(2,2)$$

$$\hat{Q}(4,1) = x_4 = a_4 x_3 + b_4$$
$$= a_4 a_3 x_2 + a_4 b_3 + b_4$$
$$= a_4 a_3 (a_2 b_1 + b_2) + (a_4 b_3 + b_4)$$
$$= a_4 a_3 \hat{Q}(2,1) + \hat{Q}(4,3)$$

In general

$$\hat{Q}(2i,1) = x_{2i} = \left(\prod_{r=i+1}^{2i} a_r \right) \hat{Q}(i,1) + \hat{Q}(2i, i+1). \quad (5)$$

Equation (5) gives us our recursive doubling. Both $\hat{Q}(i,1)$ and $\hat{Q}(2i, i+1)$ are identical in structure since they both require the same number and sequence of multiplications and additions. Also, each of these terms involves i a's and i b's, exactly one-half the number of a's and b's used in $\hat{Q}(2i,1)$. Thus if at the kth step we want to compute x_{2i}, then at the $k-1$st step we should have one processor compute $\hat{Q}(i,1)$ and another compute $\hat{Q}(2i, i+1)$. We then continue this splitting operation recursively. The resulting computation graph for the case $N = 8$ is given in Fig. 1.

Note that when we compute $\hat{Q}(2i,1)$ from the two equally complex subterms $\hat{Q}(i,1)$ and $\hat{Q}(2i, i+1)$, we also need the additional product $(\prod_{r=i+1}^{2i} a_r)$. This is not a serious hindrance since we can compute the products using the scheme shown in Fig. 2. We see that in all cases the correction products needed at one level of the tree in Fig. 1 are always available just after the previous level in Fig. 2. Figs. 1 and 2 show the computation of $\hat{Q}(8,1)$. However, it is straightforward to extend the computation to eight processors, and compute $\hat{Q}(i,1)$ for $1 \leqslant i \leqslant 8$ in parallel. The algorithm solves (3) in a time proportional to $\lceil \log_2 N \rceil$.

An example of the complete solution of (3) for the case $N = 8$ is given in detail in Table I.

B. A General Class of First-Order Recurrence Equations

In this section we define a general class of first-order recurrence equations for which we develop a parallel algorithm. The limitation to first-order equations is not as restrictive as it might first appear, since it is often the case that we can very easily reformulate a more general mth-order problem as a first-order problem. Section III-B describes such a reformulation.

The general parallel algorithm developed in Section II-C solves all recurrence equations that can be placed in the following form:

$$x_1 = b_1$$
$$x_i = f_i(x_{i-1}) = f(b_i, g(a_i, x_{i-1})), \quad 2 \leqslant i \leqslant N \quad (6)$$

where b_i and a_i are arbitrary constants and f and g are index-independent functions that satisfy the following restrictions.

Restriction 1: f is associative. $f(x, f(y,z)) = f(f(x,y), z)$.

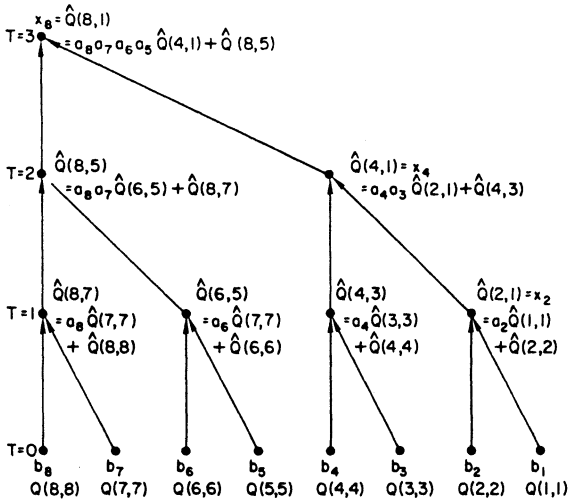


Fig. 1. Parallel computation of x_8 in the sequence $x_i = a_i x_{i-1} + b_i$.

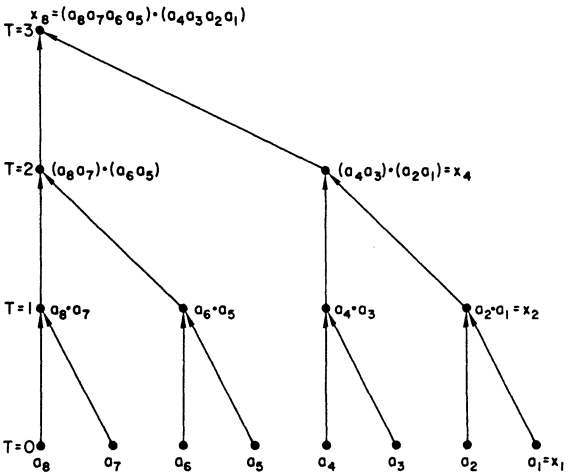


Fig. 2. Parallel computation of $x_8 = \Pi_{j=1}^{8} a_j$.

Restriction 2: g distributes over f. $g(x, f(y,z)) = f(g(x,y), g(x,z))$.

Restriction 3: g is semiassociative, that is, there exists some function h such that $g(x, g(y,z)) = g(h(x,y), z)$.

The previous restrictions on f and g are the only ones necessary to prove the correctness of the general parallel algorithm. However, these restrictions may also limit the domains from which a_i and b_i and the variables x_i can be chosen. For most normal arithmetic operators like + or · there is no problem, but more exotic operations such as floor, ceiling, modulo division, etc., may constrain the permissible domains and should be checked carefully.

The semiassociative property of g forces h to behave as if it were associative. In particular, we have

$$g(h(h(a,b),c),d) = g(h(a,b), g(c,d))$$
$$= g(a, g(b, g(c,d)))$$
$$= g(a, g(h(b,c),d))$$
$$= g(h(a, h(b,c)), d).$$

Hence, iterated compositions of h when used as the first argument of the function g can be evaluated as if h were as-

TABLE I

Processor	$T=0$ $A(i)$	$T=0$ $B(i)$	$T=1$ $A(i)$	$T=1$ $B(i)$	$T=2$ $A(i)$	$T=2$ $B(i)$	$T=3$ $A(i)$	$T=3$ $B(i)$
1	**	$b_1 = X_1 = \hat{Q}(1,1)$	**	$X_1 = \hat{Q}(1,1)$	**	$X_1 = \hat{Q}(1,1)$	**	X_1
2	a_2	$b_2 = \hat{Q}(2,2)$	a_2	$a_2x_1 + b_2 = x_2 = \hat{Q}(2,1)$	a_2	$X_2 = \hat{Q}(2,1)$	a_2	X_2
3	a_3	$b_3 = \hat{Q}(3,3)$	a_3a_2	$a_3b_2 + b_3 = \hat{Q}(3,2)$	a_3a_2	$(a_3a_2)X_1 + (a_3b_2 + b_3) = X_3 = \hat{Q}(3,1)$	a_3a_2	X_3
4	a_4	$b_4 = \hat{Q}(4,4)$	a_4a_3	$a_4b_3 + b_4 = \hat{Q}(4,3)$	$a_4a_3a_2$	$(a_4a_3)X_2 + (a_4b_3 + b_4) = X_4 = \hat{Q}(4,1)$	$a_4a_3a_2$	X_4
5	a_5	$b_5 = \hat{Q}(5,5)$	a_5a_4	$a_5b_4 + b_5 = \hat{Q}(5,4)$	$a_5a_4a_3a_2$	$a_5a_4(a_3b_2 + b_3) + a_5b_4 + b_5 = \hat{Q}(5,2)$ $= \sum_{w=2}^{5} \left(\prod_{m=w+1}^{5} a_m \right) b_w$	$\prod_{m=2}^{5} a_i^*$	X_5
6	a_6	$b_6 = \hat{Q}(6,6)$	a_6a_5	$a_6b_5 + b_6 = \hat{Q}(6,5)$	$a_6a_5a_4a_3$	$\sum_{w=3}^{6} \left(\prod_{m=w+1}^{6} a_m \right) b_w = \hat{Q}(6,3)$	$\prod_{m=2}^{6} a_i^*$	X_6
7	a_7	$b_7 = \hat{Q}(7,7)$	a_7a_6	$a_7b_6 + b_7 = \hat{Q}(7,6)$	$a_7a_6a_5a_4$	$\sum_{w=4}^{7} \left(\prod_{m=w+1}^{7} a_m \right) b_w = \hat{Q}(7,4)$	$\prod_{m=2}^{7} a_i^*$	X_7
8	a_8	$b_8 = \hat{Q}(8,8)$	a_8a_7	$a_8b_7 + b_8 = \hat{Q}(8,7)$	$a_8a_7a_6a_5$	$\sum_{w=5}^{8} \left(\prod_{m=w+1}^{8} a_m \right) b_w = \hat{Q}(8,5)$	$\prod_{m=2}^{8} a_i^*$	X_8

* Not really needed to compute $X_1, \cdots, X_8$.
** Arbitrary.

sociative without altering the output value of g. In all interesting practical problems discovered thus far, the function h is associative.

C. Parallel Algorithm

The principle of recursive doubling can be applied in a natural way to any recurrence equation that satisfies the restrictions of Section II-B. In fact, the resulting general algorithm bears a very strong resemblance to the example of Section II-A. Before giving the algorithm, however, we first give two definitions.

Definition: For any function q of two arguments define the generalized composition of q as $q_{j=n}^{(m)}(a_j)$, where

$$q_{j=n}^{(n)}(a_j) = a_n, \qquad \text{for } n \geqslant 1$$

$$q_{j=n}^{(m)}(a_j) = q(a_m, q_{j=n}^{(m-1)}(a_j)), \qquad \text{for } m > n \geqslant 1.$$

$$= q(a_m, q(a_{m-1}, \cdots, q(a_{n+2}, q(a_{n+1}, a_n)) \cdots).$$

If we let $q(a, b) = a + b$ (scalar addition), then

$$q_{j=n}^{(m)}(a_j) = (a_m + (a_{m-1} + \cdots + (a_{n+2} + (a_{n+1} + a_n)) \cdots)$$

$$= \sum_{j=n}^{m} a_j.$$

Likewise, if $q(a, b) = a \cdot b$ (scalar multiplication), then

$$q_{j=n}^{(m)}(a_j) = \prod_{j=n}^{m} a_j.$$

Definition: Define $Q(m, n)$ as

$$Q(m, n) = f_{j=n}^{(m)}(g[h_{r=j+1}^{(m)}(a_r), b_j]), \qquad m \geqslant n \geqslant 1$$

where we define

$$g(h_{r=m+1}^{(m)}(a_r), b_j) = b_j.$$

If we consider the case where f is scalar addition, and g and h are scalar multiplication, then the $Q(m, n)$ defined previously is exactly the same as the $\hat{Q}(m, n)$ defined for the example in Section II-A.

The similarities between Q and $\hat{Q}$ carry even further. The function $Q(i, 1)$ is the solution of the general recurrence equation (6), that is,

$$x_i = Q(i, 1), \qquad \forall\ 1 \leqslant i \leqslant N. \tag{7}$$

Also, as in the example, we can derive a formula computing $Q(2i, 1)$ strictly in terms of two equally complex subterms, namely,

$$Q(2i, 1) = f(Q(2i, i+1), g(h_{j=i+1}^{(2i)}(a_j), Q(i, 1))). \tag{8}$$

Both (7) and a more general version of (8) are proved in the Appendix.

Equation (8) is a perfect candidate for recursive doubling. $Q(2i, i+1)$ and $Q(i, 1)$ are identical in terms of the number of unique a's and b's referenced and require the same sequence of f, g, and h function calls to evaluate them. As with the second example, the only hindrance in implementing (8) directly as a recursive doubling algorithm is the correction term, the h composition. However, since h can be treated as an associative function, we can use a scheme similar to Fig. 2 to compute these correction terms exactly as they are needed.

Fig. 3 is a computation graph using (8) and the h composition algorithm to compute x_8. Despite its increased complexity, the general structure of this graph is identical to Figs. 1 and 2 and can be extended to solve for all elements of the sequence $x_1, \cdots, x_N$ in parallel.

We can now state the complete algorithm for solving our general recurrence equations. The detailed proof of the correctness of this algorithm is given in the Appendix.

Algorithm A–General Algorithm: This algorithm solves for $x_1, x_2, \cdots, x_N$ where $x_i = f(b_i, g(a_i, x_{i-1}))$ and f and g satisfy the restrictions of Section II-B.

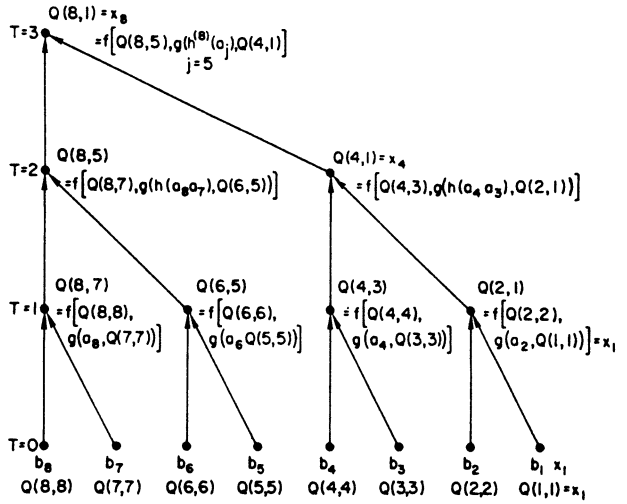


Fig. 3. Parallel computation of x_8 from the general recurrence equation.

The algorithm requires two vectors A and B of N elements. The ith component of each vector, namely $A(i)$ and $B(i)$, is stored in the memory of processor (i). The actual data structure required to represent $A(i)$ and $B(i)$ depends on the definition of the domain of the entities a_i and b_i in the basic equation (8) and may be scalars, matrices, lists, etc., depending on the problem.

Let $A^{(k)}(i)$ and $B^{(k)}(i)$ represent respectively, the contents of $A(i)$ and $B(i)$ after the kth step of the following algorithm.

Initialization Step ($k = 0$):

$B^{(0)}(i) = b_i$ for $1 \leqslant i \leqslant N$.

$A^{(0)}(i) = a_i$ for $1 < i \leqslant N$.

$A(1)$ is never referenced and may be initialized arbitrarily.

Recursion Steps: For $k = 1, 2, \cdots \lceil \log_2 N \rceil$ do each of the following assignment statements:

$$B^{(k)}(i) = f(B^{(k-1)}(i), g(A^{(k-1)}(i), B^{(k-1)}(i - 2^{k-1}))), \quad \text{for } 2^{k-1} < i \leqslant N. \quad (9)$$

$$A^{(k)}(i) = h(A^{(k-1)}(i), A^{(k-1)}(i - 2^{k-1})), \quad \text{for } 2^{k-1} + 1 < i \leqslant N. \quad (10)$$

Each statement is assumed to be evaluated simultaneously by all processors whose indices lie in the specified interval.

After the $\lceil \log_2 N \text{th} \rceil$ step, $B(i)$ contains x_i for $1 \leqslant i \leqslant N$.

End of Algorithm A.

Several things should be noted about any implementation of Algorithm A. First, when the ith processor executes (9) and (10) in that order, it must have the old values of $B(i - 2^{k-1})$ and $A(i - 2^{k-1})$, which can only be obtained from processor $(i - 2^{k-1})$. Thus at the beginning of the kth recursion step, all processors must shift their values of A and B to the processors with index 2^{k-1} greater than their own. Exactly how this data routing is performed depends on the processor interconnection pattern available in a given computer system.

Another problem with implementing Algorithm A lies in limiting the processors that execute (9) and (10) to just those with the proper indices. The masking feature (Section I-B) is the most direct way. This, however, requires executing explicit mask instructions during each recurrence step. If the number of available processors is greater than about $3N/2$, another method can avoid these extra instructions. The N processors with the highest indices are allocated to the solving of $x_1, \cdots, x_N$, and the next $N/2$ processors are initialized so that when one of the top N processors references their data, the values returned cause no change in the higher processor's values for $A(i)$ and $B(i)$. These bottom $N/2$ processors are completely masked off initially so that these initial values never change. These initial values are

$$A(i) = I, \quad \text{for } -N/2 \leqslant i \leqslant 1$$
$$B(i) = Z, \quad \text{for } -N/2 \leqslant i \leqslant 0$$

where for all a and b, $h(a, I) = a$ and $f(b, g(a, Z)) = b$. For the example of Section II-A, I is simply 1, and Z is 0.

III. Applications

A. Various First-Order Problems

As has been mentioned before, Algorithm A is applicable to a rather wide class of problems. Table II gives a collection of such problems that satisfy the functional constraints stated in earlier sections.

An interesting case occurs when we constrain all the a_i of Example 1 in Table II to be the same number z as indicated in Example 5 in Table II. We then get the recursion

$$x_i = zx_{i-1} + b_i$$

which, if we solve for x_N, yields

$$x_N = b_1 z^{N-1} + b_2 z^{N-2} + \cdots + b_{N-1} z + b_N.$$

But this is simply the evaluation of the polynomial $b_1 x^{N-1} + \cdots + b_N$ at $x = z$. In fact, Algorithm A in this case is simply the parallel evaluation of polynomials (Munro and Paterson [3]).

B. Extension to mth-Order Equations

The algorithm given in the previous sections is applicable to a class of first-order recurrence equations. However, a little manipulation of the description of a problem can often convert an mth-order recurrence equation into a first-order equation with a slightly more complicated data structure. The clue to how this is done can be found in the third example in Table II, a matrix or "state variable" problem.

As an example, consider the problem

$$x_i = a_{i,1} x_{i-1} + \cdots + a_{i,m} x_{i-m} + b_i. \quad (11)$$

We wish to reformulate it in a form amenable to Algorithm A.

The first step is to see that we can collapse the m x's that are needed in (11) into a single new "variable" by using state variable notation as follows.

Let

$$Z_i = \begin{bmatrix} x_i \\ \cdot \\ \cdot \\ \cdot \\ x_{i-m+1} \end{bmatrix}. \quad (12)$$

Now we can rewrite (11) as

TABLE II
APPLICATIONS OF ALGORITHM A

Example	Domain of b	Domain of a	Domain of x	$f(a, b)$	$g(a, b)$	$h(a, b)$	Comments
1)		real numbers		$a + b$	$a \cdot b$	$a \cdot b$	$x_{i+1} = b_{i+1} + a_{i+1} x_i$
2)		real numbers		$a \cdot b$	$b \uparrow a$	$a \cdot b$	$x_{i+1} = b_{i+1} \cdot (x_i \uparrow a_{i+1})$, "$\uparrow$" is exponentiation
3)	$m \times 1$ matrix	$m \times m$ matrix	$m \times 1$ matrix	vector addition	mult. of matrix by vector	matrix mult.	$x_{i+1} = B_{i+1} + A_{i+1} x_i$ where A is $m \times m$ and x, B are $m \times 1$
4)		real numbers		b	min (a, b)	min (a, b)	x_i is the smallest of $a_1, \cdots, a_i$
5)		real numbers		b	max (a, b)	max (a, b)	x_i is the largest of $a_1, \cdots, a_i$
6)	real number	any real number z	real number	$a + b$	$a \cdot b$	$a \cdot b$	$x_i = x_{i-1} \cdot z + b_i$, polynomial evaluation $x_N = P(z) = b_1 z^{N-1} + b_2 z^{N-2} + \cdots + b_{N-1} z + b_N$

$$Z_i = \begin{bmatrix} a_{i,1} \cdots a_{i,m} \\ 1\,0 \cdots 0 \\ 0\,1\,0 \cdots 0 \\ 0 \cdots 0\,1\,0 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ \cdot \\ \cdot \\ \cdot \\ x_{i-m} \end{bmatrix} + \begin{bmatrix} b_i \\ 0 \\ \cdot \\ \cdot \\ 0 \end{bmatrix} \tag{13}$$

$$= \hat{A}_i Z_{i-1} + \hat{B}_i \tag{14}$$

where $\hat{A}_i$ and $\hat{B}_i$ are the $m \times m$ matrix and $m \times 1$ vector respectively. The first row of $\hat{A}_i$ represents the original (11) and the remaining rows simply select the proper x_j to make Z_i be consistent.

Equation (14), however, is in exactly the right format for Example 3 of Table II to be applied. The variables in the recursion are m-element vectors, the $\hat{A}_i$ are $m \times m$ matrices, and the B_i are m-element vectors. The function f is vector addition, g is multiplication of a matrix by a vector, and h is matrix multiplication. Thus if we rewrite (11) into (14) we can apply Algorithm A to get a parallel solution to the original problem (11).

This particular formulation, however, is not very efficient in its use of the parallel processors. At the end of the calculation we have N m-element vectors $Z_1, \cdots, Z_N$. Only one mth of each Z_i, namely its first component x_i, represents new calculations not available from previous Z's. Most of the matrix calculations done in the recurrence steps are redundant.

We can increase the amount of parallelism in the problem by propagating (14) forward m steps before using Algorithm A. This results in a new formulation of the problem, which yields $Z_{(k+1)m} = (x_{(k+1)m}, \cdots, x_{km+1})'$ directly from $Z_{km} = (x_{km}, \cdots, x_{(k-1)m+1})'$.

It is easy to show by induction that Z_{km+m} can be computed as follows:

$$Z_{km+m} = \left(\prod_{j=km+1}^{km+m} \hat{A}_j \right) Z_{km} + \sum_{r=km+1}^{km+m} \left(\prod_{j=r+1}^{km+m} \hat{A}_j \right) \hat{B}_r,$$

$$k = 1, \cdots, N/m - 1. \tag{15}$$

This equation can be restated in a form directly usable by Algorithm A as follows.

Let

$$X_{k+1} = Z_{(k+1)m} = \begin{bmatrix} x_{(k+1)m} \\ \cdot \\ \cdot \\ \cdot \\ x_{km+1} \end{bmatrix}$$

$$A_{k+1} = \left(\prod_{j=km+1}^{(k+1)m} \hat{A}_j \right) \quad B_{k+1} = \sum_{r=km+1}^{(k+1)m} \left(\prod_{j=r+1}^{(k+1)m} \hat{A}_j \right) \hat{B}_r. \tag{16}$$

Now (15) becomes

$$X_{k+1} = A_{k+1} X_k + B_{k+1}, \quad k = 1, \cdots, N/m \tag{17}$$

which again is our familiar first-order linear matrix recurrence equation.

Now to compute all N elements of (11), we need only compute N/m elements of the series $X_1, \cdots, X_{N/m}$ using (17). Using Algorithm A we can compute these N/m elements with $\lceil \log_2 N/m \rceil$ applications of the recurrence step, plus some initial time to compute the initial A's and B's given by (16). Further, since there are only N/m elements to compute, Algorithm A also calls for only N/m processors.

The important aspect of this reformulation is not that the number of steps has been reduced, but that the number of processors has dropped. Equation (17) takes $\lceil \log_2 m \rceil$ fewer recurrence iterations to evaluate than does (14), but about $\lceil \log_2 m \rceil$ additional iterations are required to set up (17) from (14) with N processors. Thus we have not reduced the time to solve the problem, but we have reduced redundant computations to the point where we need only N/m processors after the initial setup.

IV. Summary and Conclusion

Various researchers have developed parallel algorithms for specific problems, such as polynomial evaluation (Munro and Paterson [3]), and the solution of tridiagonal systems of equations (Buneman [1], Buzbee *et al.* [2], and Stone [4]). As with Algorithm A, these algorithms typically require execution times proportional to $\lceil \log_2 N \rceil$. None of them, however, is applicable to any wider class of problems than the particular ones they were designed to solve. Algorithm A, on the other hand, solves any problem for which the solution can be stated in terms of a recurrence equation satisfying a few simple restrictions. It is worthwhile mentioning that the running time for Algorithm A can vary widely from problem to problem

even though the time is always proportional to $\lceil \log_2 N \rceil$. The constant of proportionality depends on the time it takes to evaluate f, g, and h. These functions can be as simple as a magnitude comparison or floating-point addition and can be as complex as a matrix multiplication, with very large differences in their respective constants of proportionality.

The power of Algorithm A comes from the generalization of the technique of recursive doubling. This technique seems to hold an important key to understanding exactly how parallelism can be extracted from what appear to be highly serial problems. The major results of this paper indicate that the class of serially stated problems that are amenable to parallel solutions is a large one, and includes some problems that have been thought to be poorly suited to parallel processors.

Appendix
Validity of Algorithm A

This Appendix contains some basic theorems that establish the validity of Algorithm A. We assume we are solving equations of the form of (6), where the functions f, g, and h all satisfy the restrictions of Section II-B. We also assume that the concept of generalized composition and the definition of the function $Q(m, n)$ carry over from Section II-C.

Theorem 1: For any i, k such that $1 \leqslant k < i \leqslant N$ then for any j such that $1 \leqslant j \leqslant k$

$$Q(i, i-k) = f(Q(i, i-j+1), g(h^{(i)}_{r=i-j+1}(a_r), Q(i-j, i-k))).$$

Proof: Assume $1 \leqslant j \leqslant k$. Then

$$f(Q(i, i-j+1), g(h^{(i)}_{r=i-j+1}(a_r), Q(i-j, i-k)))$$
$$= f(Q(i, i-j+1), g(h^{(i)}_{r=i-j+1}(a_r), f^{(i-j)}_{m=i-k}(g(h^{(i-j)}_{r=m+1}(a_r), b_m))))$$
[by definition of $Q(i-j, i-k)$]
$$= f(Q(i, i-j+1), f^{(i-j)}_{m=i-k}(g(h^{(i)}_{r=i-j+1}(a_r), g(h^{(i-j)}_{r=m+1}(a_r), b_m))))$$
[g distributes over f]
$$= f(Q(i, i-j+1), f^{(i-j)}_{m=i-k}(g(h^{(i)}_{r=m+1}(a_r), b_m)))$$
[g is semiassociative]
$$= f(f^{(i)}_{m=i-j+1}(g(h^{(i)}_{r=m+1}(a_r), b_m)), f^{(i-j)}_{m=i-k}(g(h^{(i)}_{r=m+1}(a_r), b_m)))$$
[definition of $Q(i, i-j+1)$]
$$= f^{(i)}_{m=i-k}(g(h^{(i)}_{r=m+1}(a_r), b_m))$$ [associativity of f]
$$= Q(i, i-k).$$

End of proof.

Theorem 2: For $1 \leqslant i \leqslant N$, $x_i = Q(i, 1)$.

Proof: By induction on i.

Basis Step: $i = 1$.

$$Q(1, 1) = b_1 = x_1 \quad \text{[by definition]}.$$

Induction Step: Assume $Q(j, 1) = x_j$ for $j < i$. Then

$$x_i = f(b_i, g(a_i, x_{i-1})) \quad \text{[recurrence equation (6)]}$$
$$= f(Q(i, i), g(h^{(i)}_{r=i}(a_r), Q(i-1, 1)))$$
[inductive hypothesis, definition of Q and h composition]
$$= Q(i, 1) \quad \text{[by Theorem 1]}.$$

End of proof.

We can now state a theorem that demonstrates the validity of Algorithm A.

Theorem 3: For all $1 \leqslant i \leqslant N$, $0 \leqslant k \leqslant \lceil \log_2 N \rceil$,

$$\text{a) } A^{(k)}(i) = \begin{cases} h^{(i)}_{r=2}(a_r), & 1 < i \leqslant 2^k + 1 \\ h^{(i)}_{r=i-2^k+1}(a_r), & 2^k + 1 < i \leqslant N \end{cases}$$

$$\text{b) } B^{(k)}(i) = \begin{cases} Q(i, 1), & 1 \leqslant i \leqslant 2^k \\ Q(i, i-2^k+1), & 2^k < i \leqslant N. \end{cases}$$

Proof: Directly by induction and Theorems 1 and 2. The proof of Theorem 3 is direct but tedious; we omit it here.

Using part b) of Theorem 3 we get the immediate result.

Corollary: After the $\lceil \log_2 N \rceil$th iteration of Algorithm A, $B(i)$ contains x_i for $1 \leqslant i \leqslant N$.

Thus we have shown that not only does Algorithm A compute the solution $x_1, \cdots, x_N$ to (6), but also that it terminates in exactly $\lceil \log_2 N \rceil$ iterations.

Acknowledgment

Recursive doubling solutions to the first-order problem of Section II-A were discovered independently by H. R. Downs of Systems Control, Inc., and H. Lomax of NASA Ames Research Center. Recursive doubling solutions to second-order linear recurrences have been known to J. J. Sylvester as early as 1853. The authors wish to thank D. Knuth for pointing out Sylvester's work and for several stimulating suggestions while this research was in progress and the referees for pointing out that the h function need not be associative.

After this paper had been reviewed and accepted for publication, the authors encountered the report by Trout [5], which has several similar results. His work was done independently of the work reported here and carries the research beyond the limits of this paper.

References

[1] O. Buneman, "A compact non-iterative Poisson solver," Stanford Univ. Inst. Plasma Res., Stanford, Calif., Rep. 294, 1969.

[2] B. L. Buzbee, G. H. Golub, and C. W. Nelson, "On direct methods for solving Poisson's equations," *SIAM J. Numer. Anal.*, vol. 7, pp. 627-656, Dec. 1970.

[3] I. Munro and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," in *Conf. Rec., 1971 12th Annu. Symp. Switching and Automata Theory*, IEEE Publ. 71 C 45-C, pp. 132-139.

[4] H. S. Stone, "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations," *J. Ass. Comput. Mach.*, vol. 20, pp. 27-38, Jan. 1973.

[5] H. R. G. Trout, "Parallel techniques," Dep. Comput. Sci., Univ. Illinois, Urbana, Rep. UIUCDCS-R-72-549, Oct. 1972.

An Algorithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis

VOJIN G. OKLOBDZIJA

Abstract— **A novel way of implementing the Leading Zero Detector (LZD) circuit is presented. The implementation is based on an algorithmic approach resulting in a modular and scalable circuit for any number of bits. We designed a 32 and 64 bit leading zero detector circuit in CMOS and ECL technology. The CMOS version was designed using both: logic synthesis and an algorithmic approach. The algorithmic implementation is compared with the results obtained using modern logic synthesis tools in the same 0.6 μm CMOS technology. The implementation based on an algorithmic approach showed an advantage compared to the results produced by the logic synthesis. ECL implementation of the 64 bit LZD circuit was simulated to perform in under 200 pS for nominal speed.**

I. INTRODUCTION

In any floating-point processor *normalization* is a required operation. It consists of an appropriate left shift until the first nonzero digit is in the left-most position. The amount of shift is determined by counting the number of zero digits from the left-most position until the first nonzero digit is reached. The exponents are appropriately decremented for the shift amount. The normalization is usually performed before storing the numbers in the register file (memory), commonly referred to as *post-normalization*, and referred to as *pre-normalization* before the operation is performed. In both cases, the special circuit implemented (in hardware) to detect the number of leading zeros is referred to as a *Leading Zero Detector* (LZD).

Applying a straight forward combinatorial approach in designing the LZD circuit is a rather complicated process because each bit of the result is dependent on all of the input bits, which in the case of 64 bit word, consists of 64 inputs. For example a 64 bit LZD circuit would consist of six outputs, each dependent on 64 inputs. It is obvious that such large fan-in dependencies are a problem and that the resulting circuit is likely to be complicated and slow. To design such a circuit using computer aided Boolean minimization techniques or the Karnaugh map method is cumbersome and slow and the resulting design does not exhibit any structure. An immediate solution would be to resort to the use of the logic synthesis tools and "let the tool do the job". This is perhaps the most commonly practiced approach, for any of the complex and complicated circuit of which LZD is a very good example.

Characteristic of the LZD circuit is its concise functional description. It is also very easy to describe the circuit using any of the common hardware description languages. Such a circuit naturally leads itself to the use of logic synthesis by describing the expected functionality in VHDL and simply letting the computer to do the rest.

On the other hand, we can use some intelligence in designing the circuit by attempting first to identify some common modules and impose hierarchy on the design. The LZD circuit in particular, is quite suitable for exploring possibilities of hierarchical and structural design. This yields to substantial improvement of the circuit regularity and speed, compared to straight-forward minimization. The resulting circuit performs well with low and regular fan-in and fan-out, leading to better wireability and better layout. However, LS tools have not yet reached a level of sophistication which can deal with hierarchical structures and create or impose hierarchy in the design. Their approach is to rather expand the logic in one level and optimize it via elaborate and laborious logic minimization, using hours of CPU time yet not being able to identify common modules and hidden structure. Therefore, the design presented in this paper results not only in an efficient and fast LZD, but also provides an efficiency measure of the logic synthesis tools and their limitations.

Manuscript received January 6, 1993; revised July 21, 1993. This work was supported by a minigrant from the Office of Research, University of California, Davis.

The author is with the Department of Electrical and Computer Engineering, University of California, Davis, CA 95616.

IEEE Log Number 9214288.

TABLE I
TWO BIT TRUTH TABLE FOR LZD

Pattern	Position	Valid
1X	0	yes
01	1	yes
00	X	no

II. DESIGN USING AN ALGORITHMIC APPROACH

In our approach we use the inherent hierarchy associated with the leading zero detection process and map it into a hierarchical and modular design. In order to understand this design process, let us begin by first examining only the two bit case, as shown in the Table I. The pattern on the left designate the possible two bit combination. If the left-most bit is "1" we assign "0" to the *Position* and "1" to the *Valid* bit indicating that there is *zero* distance from the left-most bit to the first *nonzero* bit. If both bits are "0", we would set *Valid* bit to 0 indicating that this is not a valid position. Not only is this because we have only one bit to indicate position, but the next two-bit group might have more zeroes to follow and therefore position information is not complete. It is straightforward to construct the logic for the two bits representing the valid bit (V) and the position bit (P) as shown in Table I.

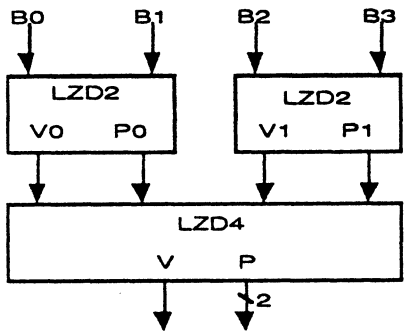
We can easily extend the two bit case to the four bit case. Let us designate the position bits (4 bits total) as $P0$ for the left-most two bits and $P1$ for the right-most two bits as shown in Fig. 1(b). Also, we will designate $V0$ and $V1$ as the valid bits for the first two and second two bits respectively starting from the left to right. The leading zero position can be represented as a function of those four bits as shown in the fifth column of Fig. 1(a) (minus sign represents complementation). Also, it should be noted that we are using "Big Endian" notation, i.e., we start indexing from left to right (this notation is used by IBM).

The 4 bit circuit has a depth of 2 logic levels; in the second level the valid bit is formed as a logical OR of the valid bits from the previous level. In other words, if there is a "valid" string of bits within the group in the previous level, then this group has a valid position bit. If all of the groups, however, do not show a "valid" output, this simply means that there are a string of zeros and that the first nonzero bit can be expected only within one of the groups to the "right". The left valid bit V_1 is inverted and concatenated with P_0 if $V_0 = 1$, or with P_1 if $V_0 = 0$ and $V_1 = 1$. This is achieved

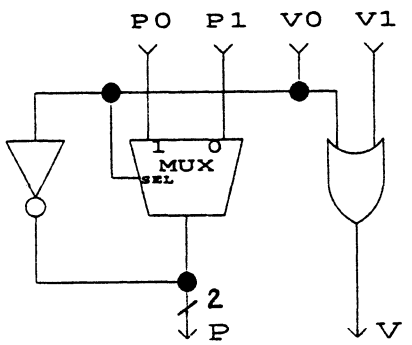
Reprinted from *IEEE Transactions on VLSI Systems*, Vol. 2, No. 1, pp. 124-128, March 1994.

pattern	position	position (binary)	valid	position
1011	0	00	yes	(-V0)P0
0100	1	01	yes	(-V0)P0
0011	2	10	yes	(-V0)P1
0001	3	11	yes	(-V0)P1
0000	X	XX	no	XX

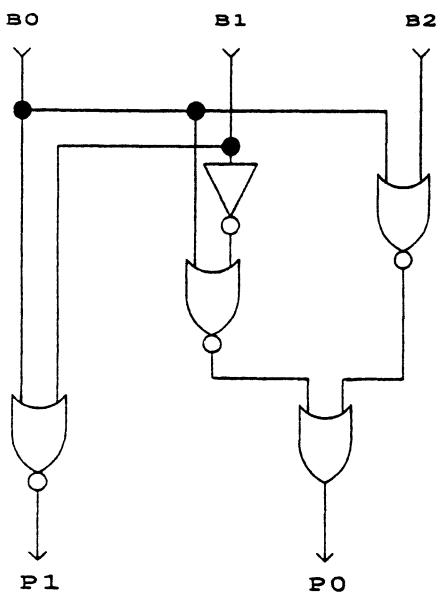
(a)



(b)



(c)



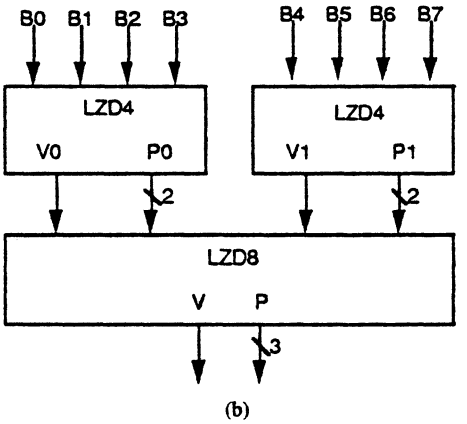
(d)

Fig. 1. Design of a 4 bit LZD. (a) Truth table. (b) Using two 2 bit LZD's. (c) The logic sturcture of the 4 bit LZD block. (d) One level implementation of the 4 bit LZD.

by simply multiplexing $P0$ and $P1$ to the output of the multiplexer. The logic structure of the 4 bit LZD group (LZD4) is shown in Fig.

8-bit LZD							
				"left" nibble		"right" nibble	
bit pattern		position	valid	P0	V0	P1	V1
1XXX	XXXX	000	yes	00	yes		
01XX	XXXX	001	yes	01	yes		
001X	XXXX	010	yes	10	yes		
0001	XXXX	011	yes	11	yes		
0000	1XXX	100	yes		no	00	yes
0000	01XX	101	yes		no	01	yes
0000	001X	110	yes		no	10	yes
0000	0001	111	yes		no	11	yes
0000	0000	XXX	no		no		no

(a)



(b)

Fig. 2. Design of an 8 bit LZD. (a) Truth table. (b) Using two 4 bit LZD's.

1(c). This implementation is particularly fast because the propagation delay of the multiplexer is smaller than the propagation delay of a gate. This is the case in several CMOS and ASIC libraries such as the ASIC library from LSI logic corporation [3].

A 4 bit LZD can also be implemented in one level directly from Fig. 1(a), avoiding the need to go into the above exercise. A one level implementation of a 4 bit LZD is shown in Fig. 1(d). It is not necessary that we limit our design to 2 bits per level. Naturally, the implementation depends on the technology used. The entire concept can be grouped in 4 bit groups. In a technology that tolerates high fan-in and fan-out we can compress even more bits into one level or one logic tree (such as in the case of ECL technology).

Now we can take two groups of 4 bits and form a LZD for an 8 bit word by simply following the same concept that we did in the example of 4 bits. The truth table is given in Fig. 2(a) and the design in Fig. 2(b).

From the above discussion we can deduce the hierarchical structure for the LZD and arrive at the following algorithm for generating the number of leading zeros:

Algorithm for generating LZ count:

(1) Form the pair of bits B_i, B_{i+1} for $i = 0$ to N-2 with bit 0 being the leftmost one

(2) Determine P and V bits for each pair

(3) for the next level determine the P_g and V_g bits as function of two pairs of inputs P and V in this level in the following way:

$V_g = V_l + V_r$ where "+" is logical OR operation of the left and right inputs

if $V_l = 1$ then $P_g = 0.P_l$ where "." designates concatenation

else if $V_r = 1$ then $P_g = 1, P_r$

othervise $V_g = 0$

Repeat step (3) $\log(N) - 2$ times

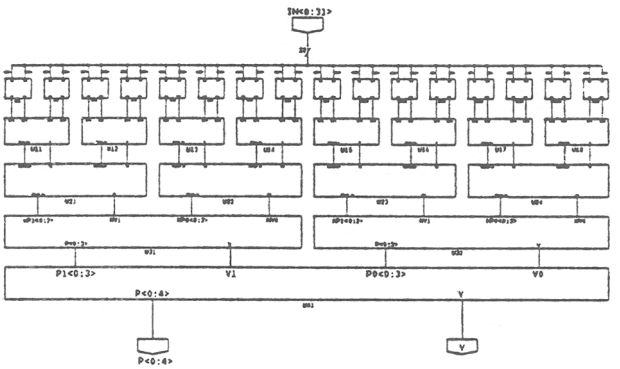


Fig. 3. 32 bit LZD circuit composed of 2 bit groups.

TABLE II
SIMULATION CONDITIONS

T_{ox}=150A, V_T=0.6V, L_{eff}=0.6u $R_{metal\ 1}$=120 mohm/sq		
NC	4.0 V, 125 C	Nominal
WC	2.8 V, 125 C	Worse Case

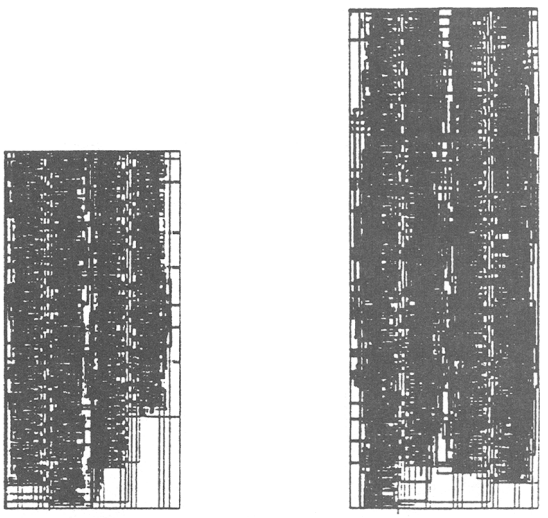
It can easily be concluded that the logical depth of this circuit is $\log_2(N)$ stages where the path through each stage is of the complexity of the multiplexer or one level of equivalent logic. The multiplexer is actually implemented using a pass-transistor structure and therefore is even faster than a regular CMOS gate. This is the reason for the extraordinary speed of this scheme for the LZD implemented in CMOS. In addition, we might want to save one level and implement this scheme in $\log(N) - 1$ levels instead of $\log(N)$ levels. This is achieved by starting with the groups of 4 bits and proceeding in the way described by the algorithm. Using CMOS technology, this is usually the best that can be achieved in terms of logic levels, because any further compression of the number of levels would pass the point of diminishing returns by increasing the fan-in and fan-out of the circuits, thus negatively affecting the speed. However, the same concept can be applied to groups of 4 bits instead of 2 bits, which is more appropriate for ECL and BiCMOS technologies. In that case there is an additional speed advantage, because the speed of this LZD implementation is proportional to $\log_4(N)$ stages and therefore faster. The structure of the 32 bit LZD composed of the 2 bit groups is shown in Fig. 3.

III. IMPLEMENTATION AND LOGIC SYNTHESIS EXPERIMENT

We implemented six LZD prototypes of various sizes. They were laid out and simulated for worst case conditions. In addition, we repeated those designs using logic synthesis tools, produced layouts and simulated the results. Our second goal was to gain performance by resizing the transistors as we went down the path. Larger transistors result in better driving capability and their size will be increased where space is available. Given that our structure is tree like, as the signal moves from the first stage to the second and third, the available space increases. By filling this space with transistors of larger size, the resulting LZD layout takes more of a rectangular shape. Such an approach is used successfully in an adder based on recurrence solving [1]. Therefore our objective was to have some performance gain by applying this findings. This provided enough data for performance analysis and evaluation of the LS tool. The technology and simulation parameters are shown in Table II.

TABLE III
PERFORMANCE OF THE NEW LZD CIRCUIT UNDER NOMINAL AND WORSE-CASE CONDITION

Bits	WC [nS]	NC [nS]
25	7.69	4.49
32	7.7	4.52
53	9.08	5.35
64	9.09	5.37
112	10.7	6.41
128	10.7	6.43



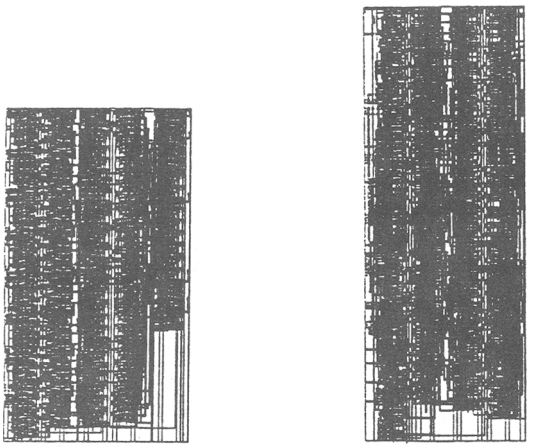
(a) (b)

Fig. 4. Layout of the 32 bit LZD. (a) for the algorithmic design and (b) LS result.

A. The Layout

The regularity of the novel LZD design can also be used to produce a more efficient layout. By creating each cell as a basic building block for each stage, the entire circuit can be routed primarily in metal lines flowing in the direction of the data-path. This results in better performance and facilitates the inclusion of the LZD in the regular data-path of the floating-point or any other unit that needs a LZD circuit.

The layout of the 32 bit LZD is shown in Fig. 4, (a) for the algorithmic design and, (b) one obtained as a result of LS. Observe that the algorithmic layout has 4 rows of cells which were placed using the Timberwolf package resulting in an area of 163 × 340 μ. The results obtained using LS have a 35% larger area resulting in 186 × 403 μm. They are plotted to the same scale and placed next to each other for easy comparison. In terms of speed, the algorithmic LZD introduces a delay of T_a = 4.5 nS, while the LZD resulting from LS has a delay of T_{ls} = 5.8 ns (both for the typical case) which is 29% slower. Therefore we can say that the algorithmically designed LZD is roughly one–third smaller and faster than the equivalent LZD resulting from logic synthesis.



(a) (b)

Fig. 5. "Rectangular" layout of the LZD circuit produced using (a) the algorithmic approach and (b) LS

We have also tried to explore the regular and hierarchical structure of this design by applying the approach described by Vuillemin and Guibas [1] to the layout of this circuit. The idea in [1] is to lay the tree-like structures like this one on a rectangular pattern in such a way that as the signal progresses down the levels, the size of the cells is made larger, increasing their driving capability so that the signal can drive more inputs in a shorter time. We implemented this idea and laid out the 32 bit LZD circuit such that the width of the circuit was kept constant. The layout of the "rectangular" LZD structures obtained using the algorithmic approach and LS are shown in Fig. 5. They are plotted on the same scale (as in Fig. 4) to point to the difference in size between the algorithmic LZD (a) and LS (b). The "rectangular" layout of the algorithmic LZD is 206×363 μm [Fig. 5(a)] while the LS resulted in 181×473 μm. The algorithmic approach has resulted in a 14.5% smaller layout compared to the LS result in this case.

B. Performance

The performance of the novel LZD was simulated under nominal and worst case conditions, denoted NC and WC, respectively. The NC and WC conditions are characterized in Table II together with the parameters typical for this CMOS process. The table shows the speed of the LZD for different sizes starting from $N = 25$ to $N = 128$ bits. This is also shown in Table I(a) for the unbuffered LZD (without the output buffers) for nominal and worst case conditions.

In terms of performance, the "rectangular" layout introduced a delay of 6.9 nS for the algorithmic LZD and 7.7 nS for the one resulting from LS (for the nominal case). The performance advantage of the algorithmic LZD was 12% over LS for the NC. For worst-case conditions, this advantage was 19%. However, when we compared the "rectangular" layout versus the regular layout in terms of performance in both cases, the results showed that overall, the "rectangular" approach did not improve the performance in every case.

The performance of the "rectangular" layout was better than the regular layout in case of 1 pF output load. The difference was 15% for the nominal case and 12% for the worst case which favors the "rectangular" layout. The performance difference is shown in Table IV(a). In the case of no load (or light load) on the output, the "rectangular" layout was worse: 10% (nominal case) and 23% (worse case). This is explained by the fact that this circuit maintains regular fan-in and fan-out, and neither one of them increases when reaching the levels closer toward the output. The performance was not affected

TABLE IV
PERFORMANCE COMPARISON OF REGULAR VERSUS RECTANGULAR 32 BIT LZD LAYOUT WITH (a) 1 pF LOAD. (b) NO LOAD CONDITION.

Speed NC (WC) for 1.0pF load [nS]	
Regular Layout (163X340μ)	Rectangular Layout (206X363μ)
7.95 (12.8)	6.9 (11.4)

(a)

Speed NC (WC) for 0pF load [nS]	
Regular Layout (163X340μ)	Rectangular Layout (206X363μ)
4.5 (7.7)	4.93 (9.5)

(b)

TABLE V
COMPARISON OF THE ALGORITHMIC LZD AND LOGIC SYNTHESIS RESULTS

Comparison of the Algorithmic LZD circuit with the LZD obtained via Logic Synthesis										
Algorithmic LZD (regular layout)					LZD obtained with Logic Synthesis					
	no load		1.0 pF load		regular	no load		rectang.	1.0 pF load	
Area[u]	WC [nS]	NC [nS]	WC [nS]	NC [nS]	Area[u]	WC [nS]	NC [nS]	Area[u]	WC [nS]	NC [nS]
163x340 86mils	7.7	4.5	12.8	7.95	186x403 116mils	12	5.8	181x473 133mils	13.6	7.7

because the input capacitance grew proportionally, as did the driving capacity of the gate. The increase in delay caused by the capacitance increase, more than offset the improved driving capability, resulting in a slight loss. Therefore, the idea [1] which worked well for a CLA-like adder structure did not work in our case. This is shown in Table IV(b). We observe that under the no-load conditions, the LZD circuit obtained via regular layout performs better. However with 1.0 pF load, the rectangular LZD outperforms the regular one in both NC and WC. This is attributed to the stronger driving capabilities of the final stages. However, we feel that just adding stronger buffers at the output nodes would still make the regular case perform better.

In Table V we compare the results for a 32 bit LZD using our approach with the one obtained using logic synthesis without load and with 1.0 pF loading capacitance on the outputs. The algorithmic LZD outperforms the one obtained via LS under all conditions. The only exception has been the marginal difference of 0.25 nS in favor of the implementation obtained by synthesis which is attributed to the rectangular layout rather than the way LZD has been implemented. With a 1.0 pF load and use of rectangular layout, algorithmic LZD is 12% faster under nominal conditions (6.9 versus 7.7 nS) and 15% faster under worst conditions (11.4 versus 13.6 nS). In any one case, we have demonstrated that our algorithmic LZD circuit outperforms LS, and that the improvement in performance ranges from 12% (NC rectangular layout, 1 pF load) to 56% (WC regular layout, no load). The improvement in the area is from 14.5% (rectangular layout) up to 35% (regular layout). This clearly demonstrates the superiority of the algorithmic approach.

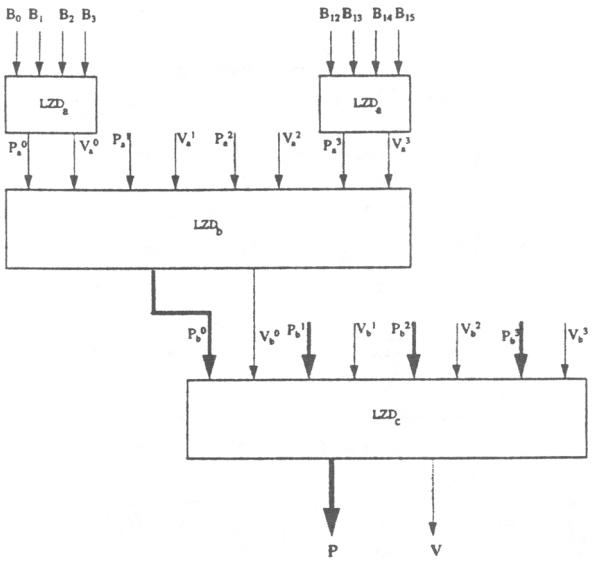


Fig. 6. Structure of 64 bit ECL LZD circuit.

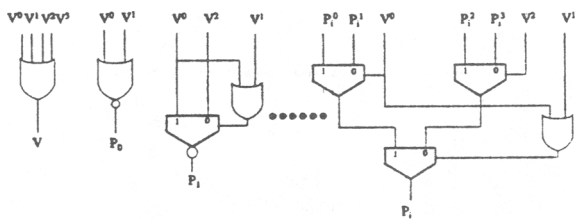


Fig. 7. ECL structure of the second level LZD circuit.

IV. HIGH-PERFORMANCE ECL IMPLEMENTATION

In a Floating-Point processor a *critical path* consists of: *leading zero detector, shift* and *rounding* operation, where each of the operations contributes approximately equally to the delay. The technology of implementation may generally differ from low power CMOS to ECL technology which is very relevant even today due to the renewed attention given to it. Implementation of LZD by the algorithm described is challenging because the rules that are applied for ECL technology are very different from those applied to CMOS. Also ECL has a tendency to combine as much logic as possible into one *level* or *logic tree,* generally allowing for larger fan-in. Finding a suitable ECL structure for any algorithmic and technology independent design is not easy. Therefore we had to solve two problems:

1) compress the design in as few levels as possible
2) identify a common and characteristic structure that also implements itself well in ECL

The first problem has been to define a 4-way LZD structure which leads to 3 levels (or 3 ECL trees) for a 64 bit LZD circuit. The first level is trivial to design and it is very much similar to the 4 bit group design shown in Fig. 1(d) except by using wired-OR (in ECL), we were able to implement everything in on the level of gates. The structure of 64 LZD implemented in ECL technology is shown in Fig. 6.

The second level will indicate the LZD position based on the input from the 4 LZD groups from the previous level. The structure of this group is not as regular as in the CMOS case. However, it was possible to identify a common multiplexer based structure that can be applied in general for any position bit ($i = 2, 3 \ldots k$). This structure is shown in Fig. 7.

Fortunately our algorithmic approach favors multiplexer structure which suits an ECL circuit very well. The bits P_0 and P_1 (as well as V) are implemented separately. The ECL tree used for this computation is shown in Fig. 8. It is three transistor levels high, which is about as much as we can implement in one ECL tree. Therefore the decision to calculate P and V based on the input of the previous four groups, rather than two or eight, seems to be optimal. The depth of the n-bit LZD circuit implemented in this way is $\log_4(n)$ levels. In our case for 64 bit LZD, we have a depth of 3 ECL *trees*. Using the advanced Motorola BiCMOS process, this circuit produces the result in $T_{cr} = 200$ pS (nominal time) for a 64 bit LZD circuit. It should be noted that the first level, which is one gate deep, could be integrated into the second level yielding an implementation in only 2 ECL trees.

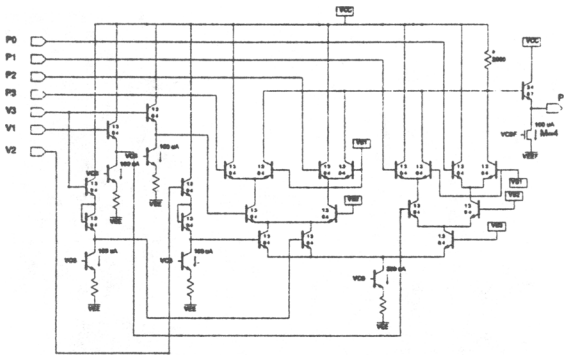


Fig. 8. ECL circuit implementation of the second level (64 bit LZD).

V. CONCLUSION

In this paper we have described an algorithmic approach to designing a leading zero detector. This circuit has been implemented in 0.6 μ CMOS technology and is compared to the results obtained using logic synthesis under various conditions and for different layout approaches. The algorithmic approach outperformed LS consistently, with improvements in speed ranging from 12%–56% and improvements in layout area ranging from 14.5%–35%. We have clearly demonstrated the superiority of the algorithmic approach on this circuit. The results generally indicate that careful analysis of the problem and clever management of the hierarchy pays big dividends in the performance of critical circuits, especially data-paths. Although very useful, LS tools are still not capable of managing hierarchies and making intelligent choices when it comes to design, and therefore they should be treated accordingly. The resulting LZD circuit has remarkable performance, which is important since it is often a part of the critical path in the floating-point unit.

ACKNOWLEDGMENT

I gratefully acknowledge V. Chang for running the simulation and Timberwolf program. I thank the referees and R. Maeder for careful reading and useful remarks. The idea was conceived of in June 1987 while the author was on the TF-1 project at IBM T. J. Watson Research Center in New York. I am grateful to M. Denneau of IBM for his ideas and discussion during the project. The author is thankful for the generous support from Sun Microsystems Laboratories, G. Taylor, D. Ditzel, and P. Hansen in particular.

REFERENCES

[1] J. Vuillemin and L. Guibas, "On fast binary addition in MOS technology," in *Proc.*, ICCC'82, New York, Sept. 28, 1982.
[2] V. G. Oklobdzija, "An implementation algorithm and design of a novel leading zero detector circuit," presented at 26th Asilomar Conf. on Signals, Systems and Computers, Oct. 26–28, 1992.
[3] LSI Logic, *1.0 micron cell-based product databook.* 1991.

SPIM: A Pipelined 64×64-bit Iterative Multiplier

MARK R. SANTORO, STUDENT MEMBER, IEEE, AND MARK A. HOROWITZ, MEMBER, IEEE

Abstract—**A 64×64-bit iterating multiplier, the Stanford Pipelined Iterative Multiplier (SPIM), is presented. The pipelined array consists of a small tree of 4:2 adders. The 4:2 tree is better suited than a Wallace tree for a VLSI implementation because it is a more regular structure. A 4:2 carry-save accumulator at the bottom of the array is used to iteratively accumulate partial products, allowing a partial array to be used, which reduces area. SPIM was fabricated in a 1.6-μm CMOS process. It has a core size of 3.8×6.5 mm and contains 41 000 transistors. The on-chip clock generator runs at an internal clock frequency of 85 MHz. The latency for a 64×64-bit fractional multiply is under 120 ns, with a pipeline rate of one multiply every 47 ns.**

I. Introduction

THE DEMAND for high-performance floating-point co-processors has created a need for high-speed, small-area multipliers. Applications such as DSP, graphics, and on-chip multipliers for processors require fast area efficient multipliers. Conventional array multipliers achieve high performance but require large amounts of silicon, while shift and add multipliers require less hardware but have low performance. Tree structures achieve even higher performance than conventional arrays but require still more area.

The goal of this project was to develop a multiplier architecture which was faster and more area efficient than a conventional array. As a test vehicle for the new architecture, a structure capable of performing the mantissa portion of a double extended precision (80 bit) floating-point multiply was chosen. The multiplier core should be small enough such that an entire floating-point co-processor, including a floating-point multiplier, divider, ALU, and register file, could be fabricated on a single chip. A core size of less than 25 mm^2 was determined to be acceptable. This paper presents a 64×64-bit pipelined array iteratively accumulating multiplier, the Stanford Pipelined Iterative Multiplier (SPIM), which can provide over twice the performance of a comparable conventional full array at one-fourth of the silicon area.

Manuscript received July 1, 1988; revised September 25, 1988 and November 21, 1988. The development of SPIM was supported in part by the Defense Advanced Project Research Agency (DARPA) under Contracts MDA903-83-C-0335 and N00014-87-K-0828.

The authors are with the Center for Integrated Systems, Stanford University, Stanford, CA 94305.

IEEE Log Number 8826243.

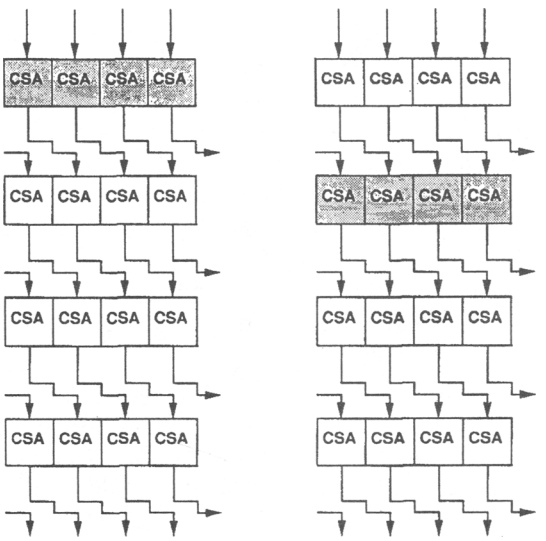


Fig. 1. Conventional array multiplier. Shaded areas represent intermediate partial product flowing down array.

II. Architectural Overview

Conventional array multipliers consist of rows of carry-save adders (CSA) where each row of CSA's sums up one additional partial product (see Fig. 1).[1] Since intermediate partial products are kept in carry-save form there is no carry propagate, so the delay is only dependent upon the depth of the array and is independent of the partial-product width. Although arrays are fast, they require a large amount of hardware which is used inefficiently. As the sum is propagated down through the array, each row of CSA's is used only once. Most of the hardware is doing no useful work at any given time. Pipelining can be used to increase hardware utilization by overlapping several calculations. Pipelining greatly increases throughput, but the added latches increase both the required hardware and the latency.

Since full arrays tend to be quite large when multiplying double or extended precision numbers, chip designers have used partial arrays and iterated using the system clock. This structure has the benefit of reducing the hardware by increasing utilization. At the limit, an iterative structure

[1]Carry-save adders are also often referred to as full adders or 3:2 adders.

Reprinted from *IEEE Journal of Solid-State Circuits*, Vol 24, No. 2, pp. 487-493, April 1989.

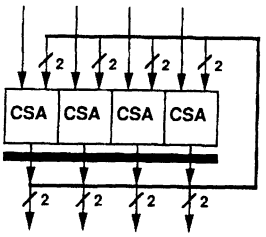


Fig. 2. Minimal iterative structure using a single row of CSA's Black bars represent latches.

would have one row of CSA's and a latch. Fig. 2 shows a minimal iterative structure. Clearly, this structure requires the least amount of hardware and has the highest utilization since each CSA is used every cycle. An important observation is that iterative structures can be made fast if the latch delays are small, and the clock is matched to the combinational delay of the CSA's. If both of these conditions are met the iterative structure approaches the same throughput and latency as the full array. This structure does, however, require very fast clocks. For a 2-μm process clocks may be in the 100-MHz range. A few companies use iterative structures in their new high-performance floating-point processors [5].

In an attempt to increase performance of the minimal iterative structure additional rows of CSA's could be added, resulting in a bigger array. For example, addition of a row of CSA cells to the minimal structure would yield a partial array with two rows of CSA's. This structure provides two advantages over the single row of CSA cells: it reduces the required clock frequency, and requires only half as many latch delays.[2] One should note, however, that although we doubled the number of CSA's, the latency was only reduced by halving the number of latch delays. The number of CSA delays remains the same. Increasing the depth of the partial array by simply adding additional rows of CSA's in a conventional structure yields only a slight performance increase. This small reduction in latency is the result of reducing the number of latches.

To increase the performance of this iterative structure we must make the CSA cells fast and, more importantly, decrease the number of series adds required to generate the product. Two well-known methods for the latter are Booth encoding and tree structures [2], [9]. Modified Booth encoding, which halves the number of series adds required, is used on most modern floating-point chips, including SPIM [7], [8]. Tree structures reduce partial products much faster than conventional methods, requiring only order $\log N$ CSA delays to reduce N partial products (see Fig. 3). Though trees are faster than conventional arrays, like conventional arrays they still require one row of CSA cells for each partial product to be retired. Unfortunately, tree structures are notoriously hard to lay out, and require large wiring channels. The additional wiring makes full trees even larger than full arrays. This has caused designers to look at permutations of the basic tree structure [1], [11].

[2] In fact one rarely finds a multiplier array that consists of only a single row of CSA's. The latch overhead in this structure is extremely high.

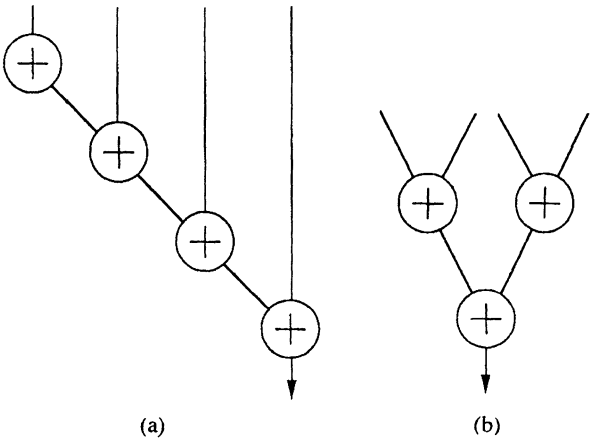


Fig. 3. (a) A conventional structure has depth proportional to N, while (b) a tree structure has depth proportional to $\log N$

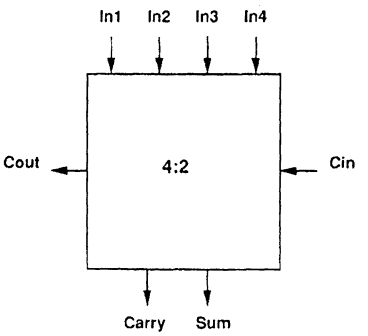


Fig 4. Block diagram of a 4:2 adder.

Unbalanced or modified trees make a compromise between conventional full arrays and full tree structures. They reduce the routing required of full trees but still require one row of CSA's for each partial product. Ideally one would want the speed benefits of the tree in a smaller and more regular structure. Since high performance was a prerequisite for SPIM, a tree structure was used. This left two problems. The first was the irregularity of commonly used tree structures. The second was the large size of the trees.

Wallace [9], Dadda [4], and most other multiplier trees use a CSA as the basic building block. The CSA takes three inputs of the same weight and produces two outputs. This 3:2 nature makes it impossible to build a completely regular tree structure using the CSA as the basic building block. A binary tree has a symmetric and regular structure. In fact, any basic building block which reduces products by a factor of 2 will yield a more regular tree than a 3:2 tree. Since a more regular tree structure was needed, the solution was to introduce a new building block: the 4:2 adder, which reduces four partial products of the same weight to 2 bits. Fig. 4 is a block diagram of the 4:2 adder. The truth table for the 4:2 adder is shown in Table I. Notice that the 4:2 adder actually has five inputs and three outputs. It is different from a 5:3 counter which takes in five inputs of the same weight and produces three outputs of different weights. The sum output of the 4:2 has weight 1 while the carry and C_{out} both have the same weight of 2. In addition, the 4:2 is not a simple counter as

TABLE I
TRUTH TABLE FOR THE 4:2 ADDER
n is number of inputs (from $In1$, $In2$, $In3$, $In4$) which = 1, C_{in} is the input carry from the C_{out} of the adjacent bit slice, C_{out} and carry both have weight 2, and sum has weight 1.

n	Cin	Cout	Carry	Sum
0	0	0	0	0
1	0	0	0	1
2	0	*	*	0
3	0	1	0	1
4	0	1	1	0
0	1	0	0	1
1	1	0	1	0
2	1	*	*	1
3	1	1	1	0
4	1	1	1	1

*Either C_{out} or Carry may be ONE for two or three inputs equal to 1 but NOT both.

C_{out} may NOT be a function of the C_{in} from the adjacent block or a ripple carry may occur.

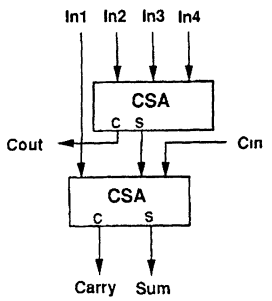


Fig. 5. A 4:2 adder implemented with two CSA's.

the C_{out} output must NOT be a function of the C_{in} input or a ripple carry could occur. As for the name, 4:2 refers to the number of inputs from one level of a tree and the number of outputs produced at the next lower level. That is, for every four inputs taken in at one level, two outputs are produced at the next lower level. This is analogous to the binary tree in which for every two inputs one output is produced at the next lower level. The 4:2 adder can be implemented directly from the truth table, or with two CSA cells as in Fig. 5.[3]

A 4:2 tree will reduce partial products at a rate of $\log_2(N/2)$ whereas a Wallace tree requires $\log_{1.5}(N/2)$, where N is the number of inputs to be reduced. Though the 4:2 tree might appear faster than the Wallace tree, the basic 4:2 cell is more complex so the speed is comparable. The 4:2 structure does, however, yield a tree which is much more regular. In addition the 4:2 adder has the advantage that two CSA's are in each pipe in place of one. This reduces both the required clock frequency and the latch overhead.

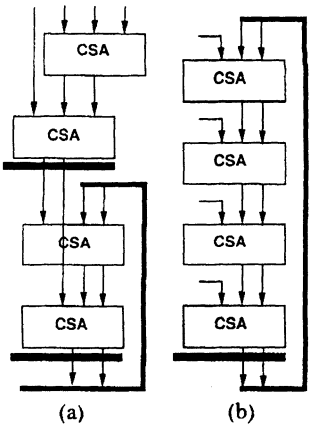


Fig. 6. With the same four CSA cells a four-input partial tree structure with a (a) carry-save accumulator will attain almost twice the throughput of a (b) partial piped array. In (a) the carry-save accumulator is placed under the 4:2 adder.

To overcome the size problem SPIM uses a partial 4:2 tree, and then iteratively accumulates partial products in a carry-save accumulator to complete the computation. The carry-save accumulator is simply a 4:2 adder with two of the inputs used to accumulate the previous outputs. The carry-save accumulator is much faster than a carry-propagate accumulator and requires only one additional pipe stage.

Fig. 6 compares a single 4:2 adder with carry-save accumulator to a conventional partial piped array.[4] Both structures reduce four partial products per cycle. Notice, however, that the tree structure is clocked at almost twice the frequency of the partial piped array. It has only two CSA cells per pipe stage, whereas the partial piped array has four. Consequently, the partial array would require 32 CSA delays to reduce 32 partial products whereas the tree structure would need only 18 CSA delays. Using the 4:2 adder with carry-save accumulator is almost twice as fast as the partial piped array, while using roughly the same amount of hardware.

The 4:2 adder structure can be used to construct larger trees, further increasing performance. In Fig. 7 we use the same 4:2 adder structure to form an eight-input tree. This allows us to reduce eight partial products per cycle. Notice that we still pipeline the tree after every two carry-save adds (each 4:2 adder). In contrast, if we clocked the tree every four carry-save adds it would double the cycle time and only decrease the required number of cycles by one. The overall effect would be a much slower multiply.

Fig. 8 shows the size and speed advantages of different sized 4:2 trees with carry-save accumulators versus conventional partial arrays. This plot is a price/performance plot where the price is size and the performance is speed (latency = 1/speed). The plot assumes we are doing a 64×64-bit multiply. Booth encoding is used, thus we must retire 32 partial products. Size has been normalized such

[3]SPIM implemented the 4:2 adder with two CSA cells because it permits a straightforward comparison with other architectures on the basis of CSA delays. By knowing the size and speed of the CSA cells in any technology, a designer can predict the size and speed advantages of this method over that currently used

[4]In Figs. 6, 7, and 9 the detailed routing has not been shown. Providing the exact detailed routing, as was done in Fig. 5, would provide more information; however, it would significantly complicate the figures and would tend to obscure their purpose, which is to show the data flow in terms of pipe stages and CSA delays.

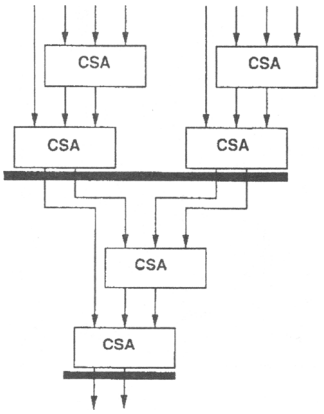


Fig. 7. An eight-input tree constructed from 4·2 adders can reduce eight partial products per cycle.

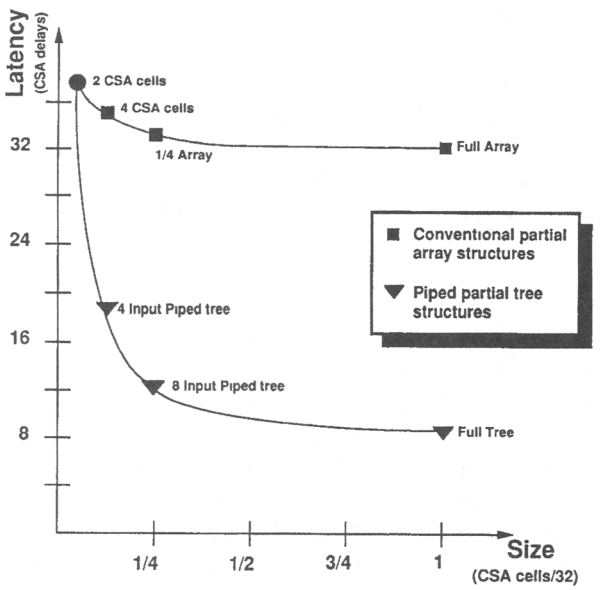


Fig. 8 Architectural comparison of piped partial tree structure with carry-save accumulator versus conventional partial array

that 32 rows of CSA cells (a full array) has a size of one unit.[5] In the upper left corner is the structure using only two rows of CSA cells. In this case the tree and conventional structures are one and the same and can be seen as a partial array two rows deep, or as a two-input partial tree. We can see that adding hardware to form larger partial arrays provides very little performance improvement. A full array is only 15 percent faster than the iterative structure using two rows of CSA's. Adding hardware in a tree-type structure, however, dramatically improves performance. For example, using a four-input tree, which uses four rows of CSA's, is almost twice as fast as the two-input tree. Using an eight-input tree is almost three times as fast as a two-input tree and only one-fourth the size of the full array.

The latency of the multiplier is determined by the depth of the partial 4:2 tree and the fraction of the partial products compressed each cycle. The latency is equal to

[5]Latency is in terms of CSA delays. We have assumed a latch is equivalent to one-third of a CSA delay in an attempt to take the latch delays into account. Size is the number of CSA cells used. It does not include the latch or wiring area.

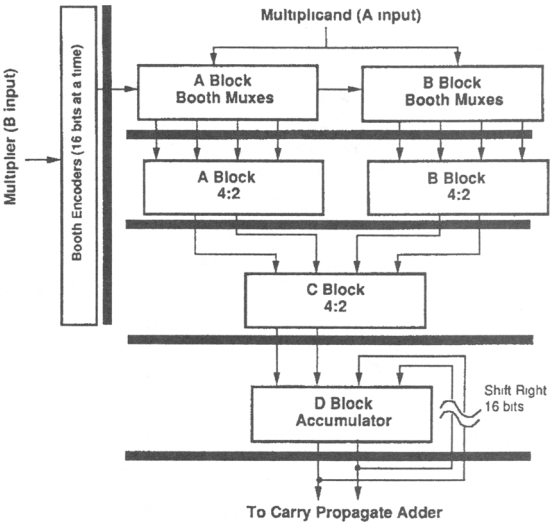


Fig. 9. SPIM data path.

$\log_2(K/2)+(N/K)$ where N is the operand size and K is the partial tree size. If Booth encoding is used N would be one-half the operand size since Booth encoding has already provided a factor of 2 compression. Start-up times and pipe stages before the tree must also be taken into account when determining latency. We choose the eight-input piped tree with Booth encoding for SPIM, as we feel this provides the best area speed trade-off for our purpose. The number of cycles required to reduce 64 bits using Booth encoding and an 8-bit tree is

$$\log_2(8/2)+(32/8)+\text{one cycle overhead} = 7 \text{ cycles}.^{6}$$

III. SPIM Implementation

Fig. 9 is a block diagram of the SPIM data path. The Booth encoders, which encode 16 bits per cycle, are to the left of the data path. The Booth-encoded bits drive the Booth select MUX's in the A and B block. The A and B block Booth select MUX outputs drive an eight-input tree structure constructed of 4:2 adders which are found in the A, B, and C blocks. Each pipe stage uses one 4:2 adder which consists of two CSA's. The D block is a carry-save accumulator. It also contains a 16-bit hard-wired right shift to align the partial sum from the previous cycle to the current partial sum to be accumulated.

Fig. 10 is a die photograph of SPIM. The A block inputs are preshifted allowing the A block to be placed on top of the B block. Using 4:2 adders in a partial tree allows the array to be efficiently routed, and laid out as a bit slice, thus making the SPIM array a very regular structure. Interestingly, the CSA cells occupy only 27 percent of the core area. The Booth select MUX's used in the A and B blocks make these blocks three times as large as the C block. Each Booth MUX with its corresponding latch is larger than a single CSA. Also, due to the routing required for the 16-bit shift, the D block is twice as large as the C block. The array area can be split into four main components: routing, CSA cells, MUX's, and latches. The routing

[6]The one-cycle overhead is used for the Booth select MUX's.

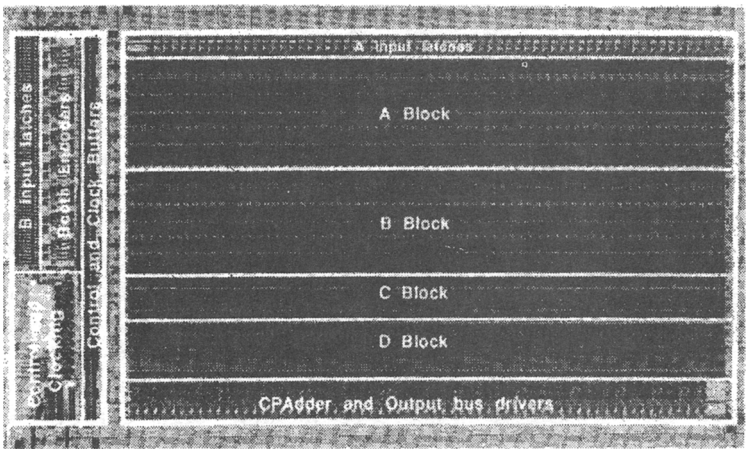


Fig. 10. Microphotograph of SPIM.

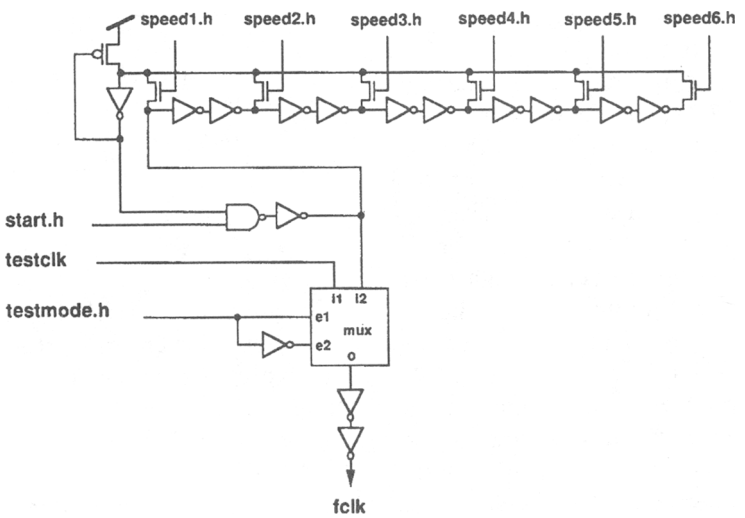


Fig. 11. SPIM clock generator circuit.

required 20 percent of the area, while the other 75 percent was equally split between the CSA cells, MUXs, and latches.

The critical path in the SPIM data path is through the *D* block. The *D* block contains the slowest path because of the added routing at the output, and the additional control MUX at its input. The input MUX is needed to reset the carry-save accumulator. It selects ZERO to reset, or the previous shifted output when accumulating. The final critical path through the D block includes two CSA cells, a master–slave latch, a control MUX, and the drive across 16 bits (128 μm) of routing.

IV. Clocking

The architecture of SPIM yields a very fast multiply; however, the speed at which the structure runs demands careful attention to clocking issues. Only two CSA's (one 4:2 adder) are found in each pipe stage, yielding clock rates on the order of 100 MHz. The typical system clock is not fast enough to be useful for this type of structure. To produce a clock of the desired frequency, SPIM uses a controllable on-chip clock generator. The clock is generated by a stoppable ring oscillator. The clock is started when a multiply is initiated, and stopped when the array portion of the multiply has been completed. The use of a stoppable clock provides two benefits. It prevents synchronization errors from occurring and it saves power as the entire array is powered down upon completing a multiply. The actual clock generator used on SPIM is shown in Fig. 11. It has a digitally selectable feedback path which provides a programmable delay element for test purposes. This allows the clock frequency to be tuned to the critical path delay. In addition, the clock generator has the ability to use an external test clock in place of the fast internally generated clock.

When a multiply signal has been received, a small delay occurs while starting up the clocks. This delay comes from two sources. The first source is the logic which decodes the run signal and starts up the ring oscillator. The second source is from the long control and clock lines running across the array. They have large capacitive loads and require large buffer chains to drive them. The simulated delay of the buffer chain and associated logic is 6 ns, almost half a clock cycle. Since the inputs are latched before the multiply is started, SPIM does the first Booth encode before the array clocks become active (cycle 0). Thus, the start-up time is not wasted. After the clocks have

TABLE II,
SPIM PIPE TIMING
Numbers indicate which partial products are being reduced. 0 is the least significant bit.

Cycle / Action	0	1	2	3	4	5	6	7
Booth Encode	startup 0-15	16-31	32-47	48-63				
A and B block Booth Muxs		0-15	16-31	32-47	48-63			
A Block CSA's			0-7	16-23	32-39	48-55		
B Block CSA's			8-15	24-31	40-47	56-63		
C Block				0-15	16-31	32-47	48-63	
D Block					clear 0-15	16-31	32-47	48-63

been started SPIM requires seven clock cycles (cycles 1–7) to complete the array portion of a multiply.

The detailed timing is shown in Table II. In the time before the clocks are started (cycle 0) the first 16 bits are Booth encoded. During cycle 1, the first 16 Booth-coded partial products from cycle 0 are latched at the input of the array. The next four cycles are needed to enter all 32 Booth-coded partial products into the array. Two additional cycles are needed to get the output through the *C* and *D* blocks. If a subsequent multiply were to follow it would have been started on cycle 4, giving a pipelined rate of four cycles per multiply. When the array portion of the multiply is complete the carry-save result is latched, and the run signal is turned OFF. Since the final partial sum from the *D* block is latched into the carry-propagate adder only every fourth cycle, several cycles are available to stop the clock without corrupting the result.

The clock generator is located in the lower left-hand side of the die (see Fig. 10). The clock signal runs up a set of matched buffers, along the side of the array, which are carefully tuned to minimize skew across the array. Wider than minimum metal lines are used on the master clock line to reduce the resistance of the clock line relative to the resistance of the driver. The clock and control lines driven from the matched buffers then run across the entire width of the array in metal.

V. Test Results

To accurately measure the internal clock frequency, the clock was made available at an output allowing an oscilloscope to be attached. SPIM was then placed in continuous (loop) mode where the clock is kept running and multiplies are piped through at a rate of one multiply every four cycles. Since the clock is continuously running its frequency can be accurately determined.

Three components determine the actual performance of SPIM: 1) the start-up time, when the clocks are started and the first Booth encode takes place (cycle 0); 2) the array time, which includes the time through the partial array plus the accumulation cycles (cycles 1–7); and 3) the carry-propagate addition (cpadd) time, when the final carry-propagate addition converts the carry-save form of the result from the accumulator to a simple binary representation. Due to limitations in our testing equipment, only the array time could be accurately measured. Since the array time requires seven cycles, and the array clock frequency was 85 MHz, the array time is simply $7 \cdot (1/85\ \text{MHz}) = 82.4$ ns. The start-up and cpadd times, based upon simulations, were 6 and 30 ns, respectively. In flowthrough mode the total latency is simply the sum of the start-up time (6 ns), the array time (82.4 ns), and the cpadd time (30 ns), for a total of 118.4 ns. Thus SPIM has a total latency under 120 ns. SPIM has a throughput of one multiply every four cycles or $4 \cdot (1/85\ \text{MHz}) = 47$ ns, for a maximum pipelined rate in excess of 20-million 80-bit floating-point multiplies per second.

The performance range of the parts tested was from 85.4 to 88.6 MHz at a room temperature of 24.5°C and a supply voltage of 4.9 V. One of the parts was tested over a temperature range of 5–100°C. At 5°C it ran at 93.3 MHz with speeds of 88.6 and 74.5 MHz at 25 and 100°C. The average power consumed at 85 MHz was 72 mA while an average of only 10 mA was consumed in standby mode.

VI. Future Improvements

The Booth select MUX's with their corresponding latches account for 38 percent of the array area. This was larger than expected. Though Booth encoding reduces the number of partial products by a factor of 2, the same result could be achieved by adding one more level of 4:2 adders to the tree. Since much of the routing already exists for the Booth MUX's, adding another level to the tree requires replacing each two Booth select MUX's with a 4:2 adder and four AND gates (see Fig. 12). Since the CSA cells are slightly larger than the Booth select MUX's the array size will grow slightly (by about 7 percent). However, if we take the whole picture into account, the core will remain about the same size, as we would no longer need the Booth encoders. Replacing the Booth encoders and Booth select MUX's with an additional level to the tree would also reduce the latency by one cycle from seven cycles to six. This occurs because the cycle required to Booth encode is now no longer needed. There are other advantages in addition to the increase in speed. Perhaps the greatest gain is the reduction in complexity. Both the Booth encoders and Booth select MUX's are now unnecessary, thus the number of cells has been reduced. In addition, Booth encoding generates negative partial products. An increase in complexity results in the need to handle the negative partial products correctly. Replacing the Booth encoders with an additional level of 4:2 adders would remove the negative partial products. Our observation is that an increase in speed and reduction in complexity can be obtained with little or no increase in area.[7]

[7]Replacing the Booth encoders and select MUX's with an additional level of 4:2 compressors is a viable alternative on more conventional, i.e., nonpiped and noniterative, trees as well. The nonpipelined speed gain depends upon the relative speed of the Booth encode plus Booth select MUX versus the delay through one 4:2 compressor and a NAND gate.

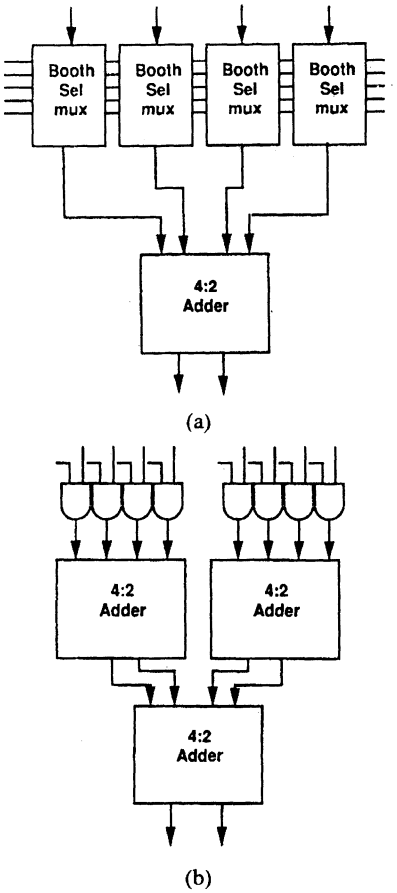


Fig. 12. Booth encoding versus additional tree level. (a) The Booth encoders and Booth select MUX's can be replaced with (b) an additional level of 4:2 adders and AND gates.

SPIM uses full static master–slave latches for testing purposes. These latches are quite large, accounting for 27 percent of the array size. In addition, they are slow, requiring 25 percent of the cycle time. Since the SPIM architecture has been proven, these latches are not required on future versions. One obvious choice is simply to replace the full static master–slave version with dynamic latches. Another option is to split the master–slave latches into two separate half latches and incorporate them into the CSA cells. This would reduce area and increase speed. A still more efficient structure is the use of single-phase dynamic latches. The balanced pipe nature of the multiplier makes the use of single-phase latches possible. Since only half as many latches are required in the pipe, single-phase dynamic latches would reduce the cycle time and decrease latch area.

Research on piped 4:2 trees and accumulators has continued. A test circuit consisting of a new clock generator and an improved 4:2 adder has been fabricated in a 0.8-μm CMOS technology. Preliminary test results have demonstrated performance in the range of 400 MHz.

VII. Conclusion

SPIM was fabricated in a 1.6-μm CMOS process through the DARPA MOSIS fabrication service. It ran at an internal clock speed of 85 MHz at room temperature. The latency for a 64×64-bit fractional multiply is under 120 ns. In piped mode SPIM can initiate a multiply every four cycles (47 ns), for a throughput in excess of 20-million multiplies per second. SPIM required an average of 72 mA at 85 MHz, and only 10 mA in standby mode. SPIM contains 41 000 transistors with a core size of 3.8×6.5 mm, and an array size of 2.9×5.3 mm.

The 4:2 adder yields a tree structure which is as efficient and far more regular than a Wallace-type tree and is therefore better suited for a VLSI implementation. By using a partial 4:2 tree with a carry-save accumulator a multiplier can be built which is both faster and smaller than a comparable conventional array. Future designs implemented in a 0.8-μm CMOS technology should be capable of clock speeds approaching 400 MHz.

Acknowledgment

Fabrication support through MOSIS is gratefully acknowledged.

References

[1] S. F. Anderson *et al.*, "The IBM system/360 model 91: Floating-point execution unit," *IBM J.*, vol. 11, no. 1, pp. 34–53, Jan. 1967.
[2] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, Part 2, 1951.
[3] J. F. Cavanagh, *Digital Computer Arithmetic Design and Implementation*. New York: McGraw-Hill, 1984.
[4] L. Dadda, "Some schemes for parallel multipliers," *Alta Freq.*, vol. 34, no. 5, pp. 349–356, Mar. 1965.
[5] B. Elkind, J. Lessert, J. Peterson, and G. Taylor, "A sub 10 ns bipolar 64 bit integer/floating point processor implemented on two circuits," in *Proc. IEEE Bipolar Circuits and Technology Meeting*, Sept. 1987, pp. 101–104.
[6] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*. New York: Wiley, 1979.
[7] P. Y. Lu *et al.*, "A 30-MFLOP 32b CMOS floating-point processor," in *ISSCC Dig. Tech. Papers*, vol. XXXI, Feb. 1988, pp. 28–29.
[8] W. McAllister and D. Zuras, "An nMOS 64b floating point chip set," in *ISSCC Dig. Tech. Papers*, Feb. 1986, pp. 34–35.
[9] C. S. Wallace, "A suggestion for fast multipliers," *IEEE Trans. Electron. Computers*, vol. EC-13, pp. 14–17, Feb. 1964.
[10] S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*. New York: CBS Publishing, 1982.
[11] D. Zuras and W. McAllister, "Balanced delay trees and combinatorial division in VLSI," *IEEE J. Solid-State Circuits*, vol. SC-21, no. 5, pp. 814–819, Oct. 1986.

A 54 × 54-b Regularly Structured Tree Multiplier

Gensuke Goto, Tomio Sato, Masao Nakajima, and Takao Sukemura

Abstract—**A 54 × 54-b parallel multiplier is implemented in 0.8-μm CMOS using the new, regularly structured tree (RST) design approach. The circuit is basically a Wallace tree, but the tree and the set of partial-product-bit generators are combined into a recurring block which generates seven partial-product bits and compresses them to a pair of bits for the sum and carry signals. This block is used repeatedly to construct an RST block in which even wiring among blocks included in wire shifters is designed as recurring units. By using recurring wire shifters, we can expand the level of repeated blocks to cover the entire adder tree, which simplifies the complicated Wallace tree wiring scheme. In addition to design time savings, layout density is increased by 70% to 6400 transistors/mm^2, and the multiplication time is decreased by 30% to 13 ns.**

I. Introduction

RAPID progress in VLSI technology has enabled the speed and performance of computers to be increased by a factor of 10 every 5 years. In addition to improvements in device technology, these advances are enhanced by improvements in processor architecture. Third-generation 32-b microprocessors integrate a floating-point processing unit on the same die as the integer unit [1], [2]. For double-precision multiplication, however, the multiplier unit is not fully implemented in hardware but is operated repeatedly for one operation. This scheme results in multiplication which is time consuming as compared to other arithmetic operations.

Further improvement in arithmetic operations will require a fresh approach. A microprocessor based on the very long instruction word (VLIW) concept [3] is a candidate for the upcoming fourth- or fifth-generation architectures. With VLIW, several arithmetic units are implemented on a single chip for parallel operation. Small high-performance units are expected to be in great demand for enhanced-performance processing systems. An improved multiplier unit is essential because the multiplier is the limiting factor in both the performance and die size of most chips today.

In the conventional Wallace tree multiplier construction [4], multi-input partial-product bits, at the same bit position, are consecutively compressed to a final sum and carry signal pair by using a series of single-bit full adders. This reduction process differs at each bit position because of the variety in the number of the partial products to be compressed. A very complicated design procedure is thus required. Using 4–2 compressors rather than full adders does simplify the design, but 100 000 manually routed interconnections are still required to design a 27 × 53-b multiplier tree [5].

An array-type multiplier consists of an array of units with full adders (3–2 compressors) and partial-product-bit generators, where each of the units processes single-bit data consecutively [6]. The entire array can be regularly laid out with only a few unit circuits, making the design very easy. Although most automatically generated multipliers are of this type [7], they are too slow for application to high-speed systems. An array and tree hybrid design may result in a multiplier of intermediate speed and complexity [8], but the difficulties in design would not be reduced much since a holding procedure is necessary to minimize the waste area.

II. Regularly Structured Tree (RST) Concepts

We propose a simple design method for a high-speed 54 × 54-b multiplier [9]. This multiplier is used for the mantissa multiplication of two double-precision numbers as outlined in the IEEE standard [10], where the mantissa of a double-precision number is represented by 52 b, and there are a hidden bit and a sign bit which are used for two's complement operations in Booth's algorithm [11]. A maximum number of 28 b may be added at the same bit position. Of these, 27 come from the partial-product-bit generators and one is from the sign bit of second-order Booth encoding.

To simplify the design process of the multiplier adder tree, we divide the tree into subcircuit modules that are reused in the construction of the complete tree. This approach has already been tried. For instance, Hokenek *et al.* [12] adopted a 7/3 counter (7 input bits reduced to a 3-b binary sum), reducing the connections among counters to half that of a tree of full adders. Even in this approach, however, time-consuming wiring design must be done on individual modules for optimization. The same is true for methods using 4–2 compressors [13], [14].

Therefore, a simplified process must contain a wiring scheme that is repeated among modules. With this in mind, we divide the adder tree and partial-product-bit generators into two equivalent blocks, 14D, to extract the maximum identity as shown in Fig. 1. The blocks are halved again to yield the 7D subblocks which have 7 input data bits. The subblocks consist of seven partial-product-bit generators and a 7–2 compressor. As they are, the sub-

Manuscript received November 26, 1991; revised May 11, 1992.

G. Goto and T. Sato are with Fujitsu Laboratories Ltd., 10-1 Morinosato-Wakamiya, Atsugi 243-01, Japan.

M. Nakajima and T. Sukemura are with Fujitsu Limited, 1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan.

IEEE Log Number 9202179.

Reprinted from *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 9, pp. 1229–1235, September 1992.

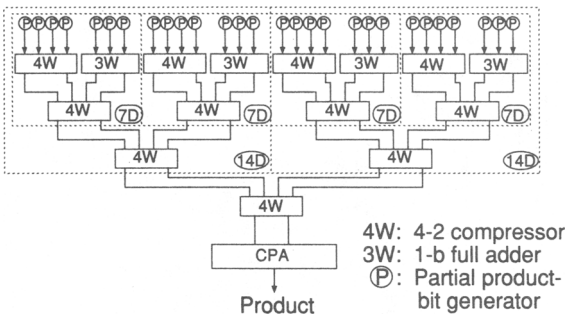


Fig. 1. Division scheme of partial-product-bit generators and adders in the 54 × 54-b tree multiplier.

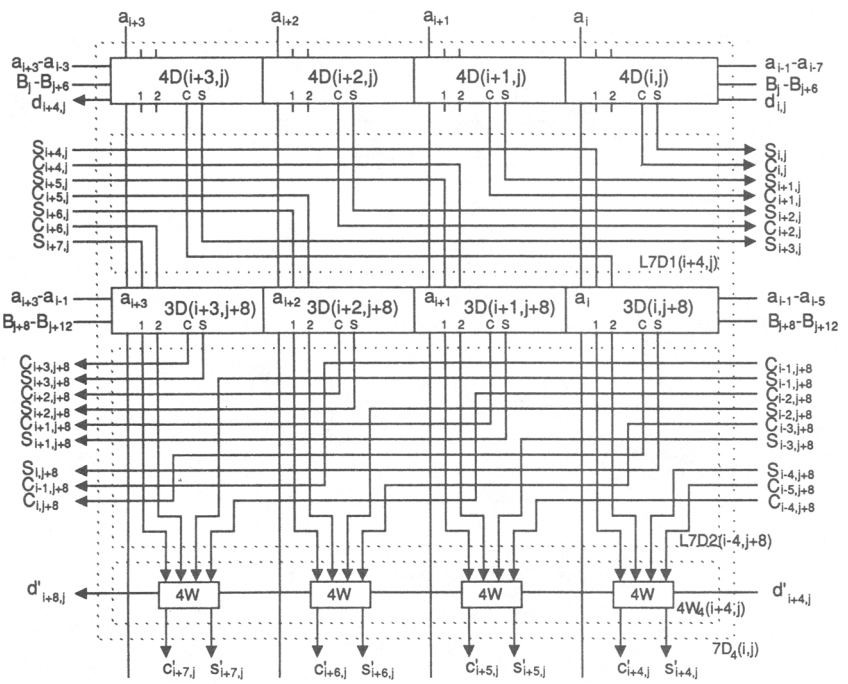


Fig. 2. Construction of the $7D_4$ (i, j) block.

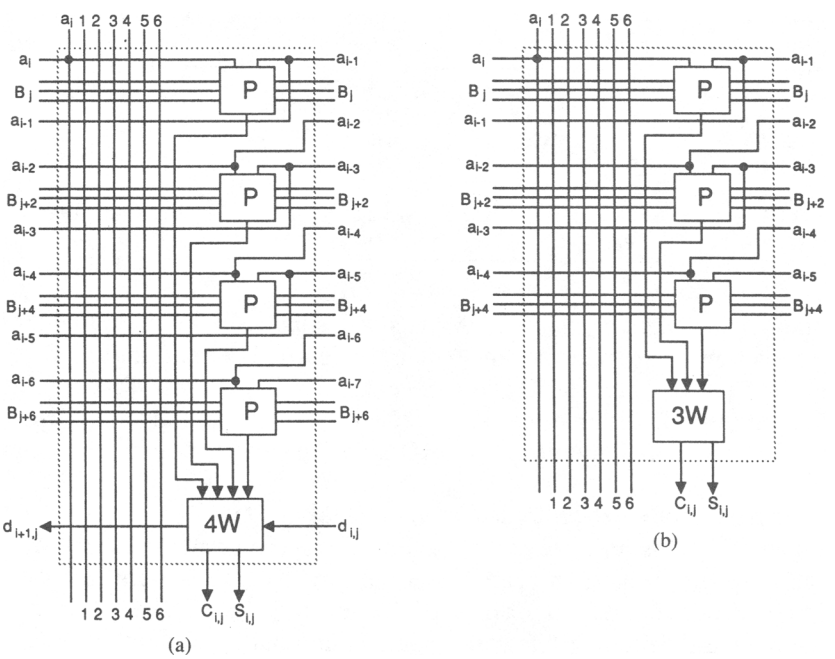


Fig. 3. Construction of (a) 4D (i, j) unit and (b) 3D (i, j) unit.

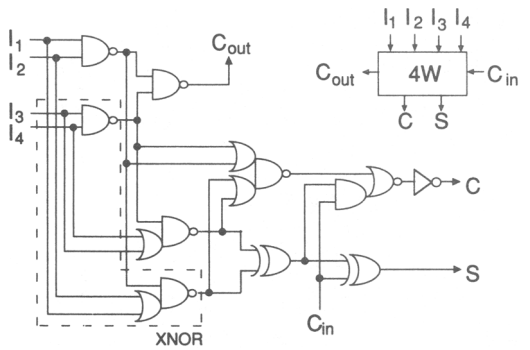


Fig. 4. Logic diagram of the 4W unit.

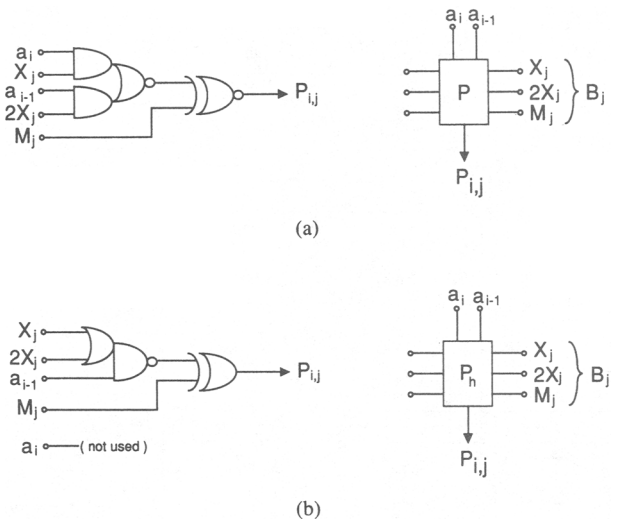


Fig. 5. Logic diagram of (a) P unit and (b) P_h unit.

TABLE I
MODIFICATIONS TO $7D_4$ BLOCK

	4D Unit				3D Unit			
Block	$(i+3, j)$	$(i+2, j)$	$(i+1, j)$	(i, j)	$(i+3, j+8)$	$(i+2, j+8)$	$(i+1, j+8)$	$(i, j+8)$
$7D_4$	4D	4D	4D	4D	3D	3D	3D	3D
$7D_{A4}$	$4D_F$	$4D_G$	4D	4D	$3D_I$	$3D_H$	3D	3D
$7D_{B4}$	$4D_F$	$4D_G$	4D	4D	$3D_F$	$3D_G$	$3D_B$	$3D_A$
$7D_{C4}$	4D	4D	4D	4D	$3D_B$	$3D_A$	$3D_B$	$3D_A$
$7D_{D4}$	4D	4D	$4D_E$	$4D_E$	3D	3D	3D	3D
$7D_{E4}$	4D	4D	$4D_E$	$4D_E$	$3D_B$	$3D_A$	$3D_B$	$3D_B$
$7D_{F4}$	$4D_D$	$4D_C$	$4D_B$	$4D_A$	$3D_A$	$3D_E$	$3D_C$	$3D_D$

blocks are not repeatable, although a packed block of four contiguous subblocks in the direction of the multiplicand bit can be used repeatedly. This packed block, $7D_4$, is shown in Fig. 2. $7D_4$ is made up of three different units: a 4D (Fig. 3(a)), a 3D (Fig. 3(b)), and a 4–2 compressor, 4W (Fig. 4).

The 4D unit consists of four partial-product-bit generators (P in Fig. 5(a)) and a 4W unit. This unit generates four partial-product bits at the same bit position and compresses them into a pair of first-level sum and carry signals. The 3D unit generates three partial-product bits and compresses them into another pair of sum and carry signals by using a single-bit full adder (3W). Output signal pairs are shifted to the right or left to generate the second-level signal pairs for the 4W units.

Two wiring pattern units are used: one for shifting the 4D signal pair to the right four bits and one to shift the 3D signal to the left four bits. The $7D_4$ block and its modified blocks, $7D_{A4}$–$7D_{F4}$, are listed in Table I. These blocks form the combined array of an adder tree and the partial-product-bit generator units.

To construct the entire 54 × 54-b multiplier circuit, some additional blocks are necessary as shown in Fig. 6, which outlines the design of our regularly structured tree

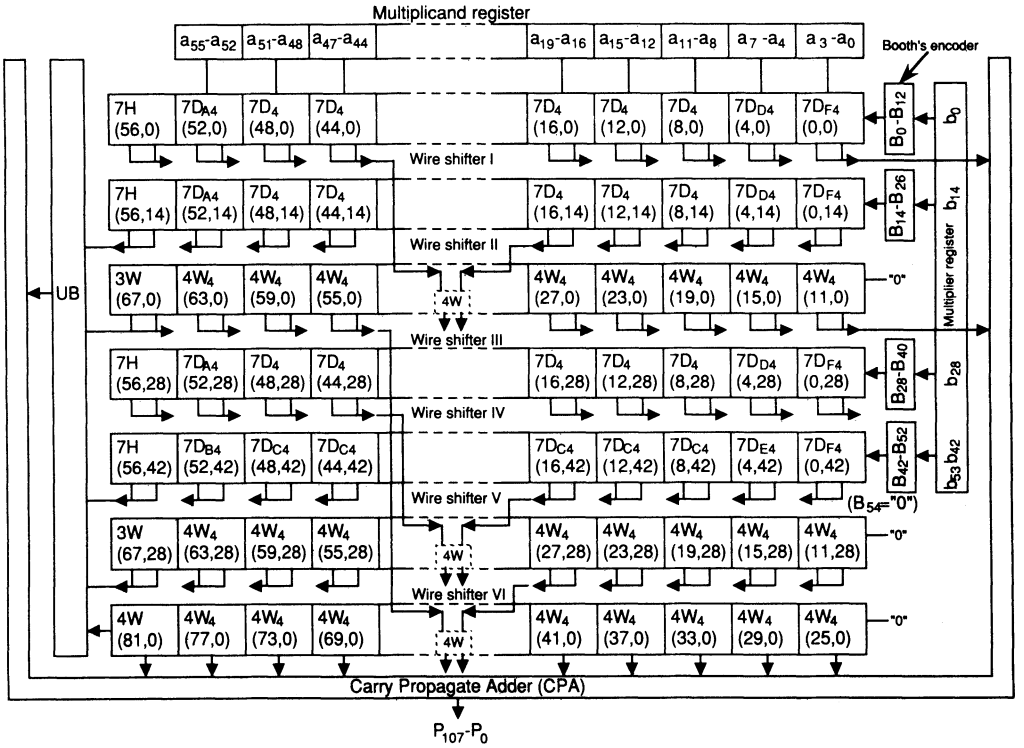


Fig. 6. Organization of regularly structured tree (RST) multiplier.

(RST) multiplier. This approach reduces multiplier design time since most blocks are repeatable in highly regular patterns which include the wiring among blocks.

III. Repeatable Block and Unit Design

Fig. 4 shows the 4–2 compressor (4W). Our full CMOS circuit uses 60 transistors. Although the gate count is the same as in the single-bit full adders in a conventional four-input Wallace tree, the speed is increased by 33%. This is because conventional adders have four EXCLUSIVE-OR (XOR) gate delays for the sum signal of four binary numbers, while 4W gives the sum in three XOR-gate delays. This may be explained by considering the outlined section in Fig. 4 as an XNOR gate whose speed is equivalent to that of an XOR gate. Simulations of the 4W with a fan-out of 4 give a delay time of 1.0 ns, in contrast to the conventional circuit delay of 1.6 ns. This verifies the above assumption.

Use of n-channel pass transistors in XOR circuits [14] may decrease the gate count to 58, but would necessitate keeping the input node wiring as short as possible. This is because the pass transistor's large resistance and the wiring capacitance distort input signals and delay the output. In most practical situations, these effects cannot be reduced without adopting buffering gates.

Fig. 5(a) shows the partial-product-bit generator (P unit) logic which is based on the second-order Booth algorithm. X_j, $2X_j$, and M_j indicate Booth encoder output for a term $b_j + b_{j-1} - 2b_{j+1}$. They represent $\times 1$, $\times 2$, and the negation of the partial product P_j for the jth bit of the multiplier. a_i and a_{i-1} are the multiplicand bits. Fig. 5(b) shows the sign bit generator (P_h unit) for the partial product. The P_h unit corrects the sign bit without expanding it to the most significant (107th) bit [15].

The construction of the $7D_4$ block must be modified for $7D_{A4}$, $7D_{B4}$, $7D_{C4}$, $7D_{D4}$, $7D_{E4}$, and $7D_{F4}$ as listed in Table I, to complete an RST block in Fig. 6. $7D_4$ is modified for $7D_{A4}$ by replacing 4D $(i + 2, j)$ and 4D $(i + 3, j)$ with $4D_G$ $(i + 2, j)$ and $4D_F$ $(i + 3, j)$. Likewise, 3D $(i + 2, j + 8)$ and 3D $(i + 3, j + 8)$ must also be replaced with $3D_H$ $(i + 2, j + 8)$ and $3D_I$ $(i + 3, j + 8)$. Modifications to other blocks may be determined similarly using Table I. Table II lists the modifications of the 4D unit for input signal variation to the 4W unit contained in the 4D unit. Table III similarly lists modifications for 3D input variation. These unit modifications are needed to fit input signals to units where there is no corresponding input on the lower and upper part of the RST—where the corrected sign bit is generated and where $+1$ is added if the Booth encoder outputs a negative partial product. The modifying process is not as cumbersome as it first appears, and the regular structure of the RST multiplier is retained.

Once $7D_4$ and the modified blocks are obtained, we can proceed with the wire shifter design as shown in Fig. 6. To complete the RST structure, we designed four kinds of wire shifters. The first kind of shifter (shifters I and IV) is for connecting second-level paired sum and carry signals output from the first (third) $7D_4$ row to the first (second) $4W_4$ row as shown in Fig. 6. This kind of shifter moves the sum signals to the right 7 b and the carry signals by 6 b, and connects the signal lines to the through-wire pairs numbered 3 and 4 in Fig. 3 in the 4D and 3D units which belong to the second (fourth) $7D_4$ row.

TABLE II
MODIFICATIONS TO 4D UNIT

Unit	Input to 4W			
	(i, j)	$(i-2, j+2)$	$(i-4, j+4)$	$(i-6, j+6)$
4D	P	P	P	P
$4D_A$	P	M_j	0	0
$4D_B$	P	0	0	0
$4D_C$	P	P	M_{j+2}	0
$4D_D$	P	P	0	0
$4D_E$	P	P	P	0
$4D_F$	0	P	P	P
$4D_G$	P_h	P	P	P

TABLE III
MODIFICATIONS TO 3D UNIT

Unit	Input to 4W		
	(i, j)	$(i-2, j+2)$	$(i-4, j+4)$
3D	P	P	P
$3D_A$	P	P	0
$3D_B$	P	P	1
$3D_C$	P	0	0
$3D_D$	P	M_j	0
$3D_E$	P	P	M_{j+2}
$3D_F$	0	P	1
$3D_G$	P_h	P	0
$3D_H$	P_h	P	P
$3D_I$	0	P	P

The second kind of shifter (shifters II and V) shifts the second-level sum signal from the second (fourth) $7D_4$ row to the left 7 b and the carry signal to the left by 8 b. Thus two sets of paired signals are added to compress the third-level paired sum and carry signals output from the first (second) $4W_4$ row.

The third type of shifter (shifter III) shifts the third-level sum signal from the first $4W_4$ row to the right 14 b and the carry signal by 13 b, and then connects to the through-wire pairs 5 and 6 in the 4D and 3D units of the third and fourth $7D_4$ row.

The fourth kind of shifter (shifter VI) shifts similar signals from the second $4W_4$ row to the left 14 and 15 b. The shifted third-level paired signals are compressed in the third $4W_4$ row to yield the fourth-level paired signals for each bit, and the output signals are input to the carry-propagate adder (CPA) for final product generation.

Wire shifters I, II, IV, and V are constructed of repeatable patterns which deal with two adjacent $7D_4$ outputs. Similarly, wire shifters III and VI are constructed of repeatable patterns over four $7D_4$ blocks. Since all the shifters form repeatable patterns, there is a significant saving in wiring design effort.

The 7H block on the left edge of the RST structure in Fig. 6 generates the second-level sum and carry signals $S'_{i+4,j}$ to $S'_{i+10,j}$, $C'_{i+4,j}$ to $C'_{i+9,j}$, and the first-level signals $S_{i,j}$ to $S_{i+3,j}$ and $C_{i,j}$ to $C_{i+2,j}$ by a combination of P, P_h, 4W, 3W, and 2W (half adder) units.

The UB block in Fig. 6 is constructed of 37 2W's and two 3W's. The UB receives these second-level signals and those from 3W(67, 28) and 4W(81, 0) to generate the third-level signals $S''_{68,0}$ to $S''_{80,0}$, $C''_{68,0}$ to $C''_{80,0}$, $S''_{97,0}$ to $S''_{107,0}$, $C''_{97,0}$ to $C''_{107,0}$, and the fourth-level signals $S'''_{82,0}$ to $S'''_{96,0}$, $C'''_{82,0}$ to $C'''_{96,0}$. All output signals beyond the 81st bit position are input to the CPA to generate final products. Output signals at lower bit positions are returned to the RST. Each of the upper thirteen 2W blocks in the UB block of Fig. 6 has the same layout pattern. The same is true for the lower twenty-four 2W blocks. The area occupied by the UB block is only a fraction of the entire multiplier area.

IV. EVALUATION

The critical path of the RST multiplier consists of four 4–2 compressors at each level which correspond to 12 XOR gate delays, plus a delay to generate a partial-product bit. This delay is equivalent to results reported by Mori *et al.* [14], which seem to be the best among reported circuits.

The speed of the 108-b CPA significantly influences multiplier performance. The CPA adopted in our design consists of a Manchester adder with a new bypass and adding scheme which yields a faster and smaller adder [16] than the conventional carry select adder. Assuming a 125% delay of 0.8-μm full-CMOS technology, the estimated delay is 8.5 ns.

The 54 × 54-b RST multiplier was fabricated on a test chip using triple-level-metal, single-polysilicon, and 0.8-μm CMOS. Both the n- and p-channel transistors have 0.8-μm gates, and the metal pitch is 2.5 μm for the first level, 3.1 μm for the second, and 5.0 μm for the third. Fig. 7 is a photomicrograph of the test chip which includes the multiplier and several sets of registers for monitoring delay time.

The wiring patterns in the repeatable blocks such as $7D_4$, $4W_4$, and all the wire shifters are implemented using the first- and the second-level metal lines. The third-level metal lines are used for multiplicand bit routing over the repeatable blocks, which run vertically on the die to minimize wiring capacitance. They are also used to supplement the power supply lines. Without the third-level wires, the die area would be 10% larger.

The rows at the top and the bottom are data registers. The RST multiplier is located in the narrow central strip. Data flow from the registers on the top and go down to the CSA (adjacent to the output registers) on the bottom edge. The multiplier measures 3.36 × 3.85 mm. There are 82 500 transistors.

Fig. 8 shows the shmoo plot at room temperature. About 20 000 random patterns and 12 specific patterns were used for this measurement. For specific patterns, the critical patterns for the RST part, the CPA, and the combined ones are included. The most critical pattern and its related output node were detected by scanning patterns. We measured the multiplier response with respect to this critical pattern and output node, using an electron beam tester (Fig. 9). The critical path delay was 13.0 ns for room temperature with a 5-V supply.

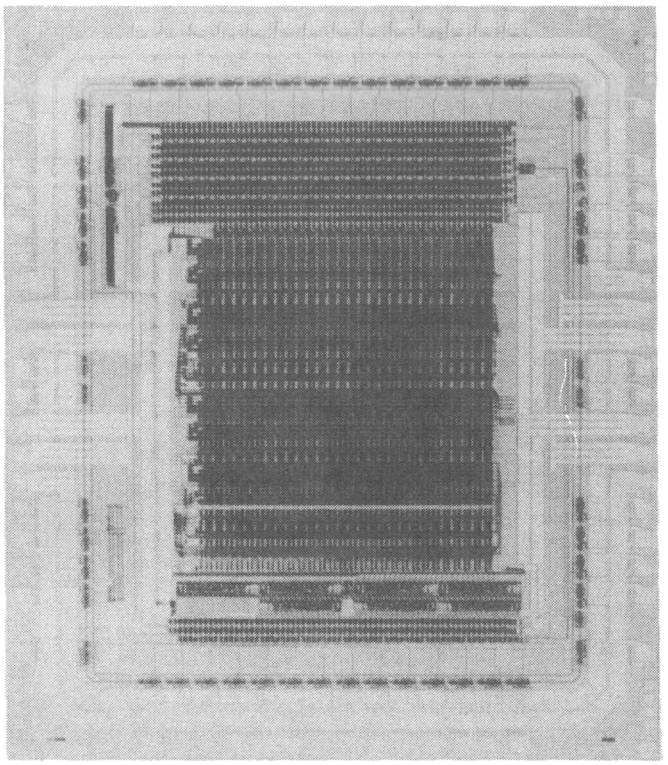


Fig. 7. Photograph of the RST multiplier chip.

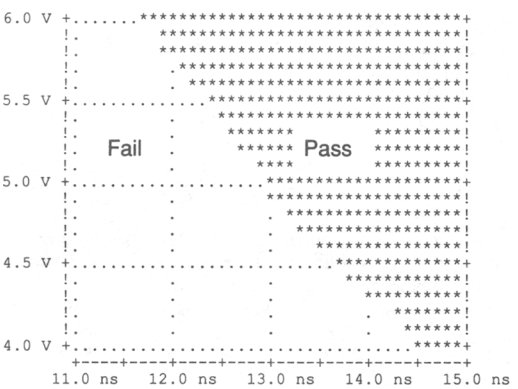


Fig. 8. Shmoo plot.

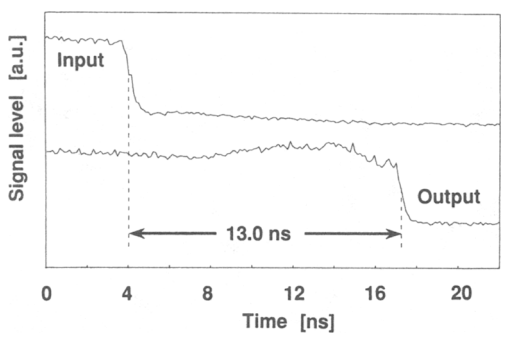


Fig. 9. Waveform of electron-beam testing on an RST multiplier.

Table IV summarizes the multiplier's features. We will now present a brief comparison of the RST multiplier with others. The 32-b multiplier in [13] uses almost the same technology as ours, but the delay time of our multiplier is

TABLE IV
RST MULTIPLIER FEATURES

Multiplier and multiplicand	54 b (including sign)
Product	107 b (including sign)
Multiplication time	13.0 ns
Power at 40 MHz	875 mW
Block size	3.36 × 3.85 mm
Transistors	82 500
Process technology	0.8-μm CMOS triple metal

30% less and the density 70% higher in a comparison on a 54-b basis. The 54-b multiplier in [14] employed 0.5-μm process technology, so its density should be far higher than the density we obtained using 0.8-μm technology. The densities, however, are almost the same. It has a smaller delay because it uses advanced 0.5-μm transistors and pseudo-CMOS logic to enhance the CPA speed. There is, however, a sacrifice in standby power consumption. Since our multiplier uses fully static CMOS gates, it does not exhibit this problem.

An analysis using shrunk 0.5-μm CMOS devices with 3.3-V power supplies produced a 54 × 54-b RST multiplier with an active area of 8.1 mm^2 and a 10-ns delay. The area is 36% smaller and the speed is comparable to the multiplier in [14], which is the smallest and fastest multiplier for double-precision numbers reported to date. The speed may be further improved by optimizing the circuits using 0.5-μm technology.

We define the quality factor Q for multiplier circuit evaluation considering the performance/cost ratio. The performance is inversely related to delay time T which is normalized by $L \cdot \log (N/2)$, where L is the gate length of the FET used, and N is the bit length to be processed. The log term is derived when using the second-order Booth algorithm and Wallace tree in the array [8] and a carry lookahead adder for the CPA [17]. The cost is proportional to the occupied area S which is normalized by $L \cdot N^2$. Thus, the Q_{54} quality factor (normalized to $N = 54$) is defined by

$$Q_{54} = (L^2 \cdot N^2 \cdot \log [N/2])/ (T \cdot S \cdot 4.17 \times 10^3) \quad (\mathrm{s}^{-1}).$$

The factors are plotted in Fig. 10 for various multipliers reported in the literature. The factor increases a little with N as a result of circuit improvement of larger multipliers. The factors of most multipliers reported to date are below 2.0. In contrast, the RST factor is 3.8—well above ratings in other designs. The RST multiplier has an excellent performance/cost ratio.

The regular pattern of the RST multiplier allows us to implement faster and denser designs than achieved by other approaches. The design time was reduced to less than one-fourth (from three months to three weeks) that required for conventional Wallace tree design because little effort was required to wire the 3W, 4W, P, and P_h circuit units, and because optimization and verification of

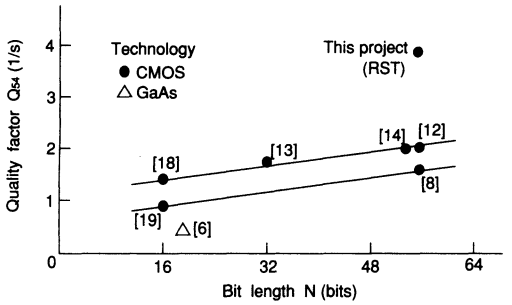


Fig. 10. Quality factors of various multipliers. The numbers in the brackets are references.

the tree circuits was much easier due to the recurring structure in the RST block.

V. Conclusion

Because of the highly regular structure of the RST multiplier, a tree multiplier with a logical hierarchy can now be laid out easily with tightly coupled recurring blocks. As a result, the tree multiplier allows a quick design which achieves the fastest and densest architecture to date. It is flexible in the sense that the repeatable $7D_4$ block can be replaced by another block to generate and compress signals of different numbers from the one presented in this paper.

The common conception that the tree multiplier is fast but large and difficult to design is no longer tenable as it is now one of the most promising approaches for implementing high-speed arithmetic units. The architecture is especially suitable for future single-chip supercomputers with strong parallel processing capability.

As this approach is systematic, a module generator based on the RST multiplier will produce multiplier designs quickly, with quality compatible to designs optimized manually.

Acknowledgment

The authors thank H. Okada and M. Sakate for their CPA design contributions, and K. Fujita and M. Kimura for their useful suggestions. We also thank H. Ishikawa, S. Hazama, and S. Mori for their guidance and support throughout this study.

References

[1] R. W. Edenfield *et al.* "The 68040 processor, part 1," *IEEE Micro*, pp. 66-78, Feb. 1990.

[2] J. H. Crawford, "The i486 CPU: Executing instructions in one clock cycle," *IEEE Micro*, pp. 27-36, Feb. 1990.

[3] J. Labrousse and G. A. Slavenburg, "A 500 MHz microprocessor with a very long instruction word architecture," in *ISSCC Dig. Tech. Papers*, Feb. 1990, pp. 44-45.

[4] C. S. Wallace, "A suggestion for fast multipliers," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 14-17, Feb. 1964.

[5] L. Kohn and S.-W. Fu, "A 1,000,000 transistor microprocessor," in *ISSCC Dig. Tech. Papers*, Feb. 1989, pp. 54-55.

[6] H. P. Singh *et al.*, "A 6.5-ns GaAs 20 × 20-b parallel multiplier with 67-ps gate delay," *IEEE J. Solid-State Circuits*, vol. 25, pp. 1226-1231, Oct. 1990.

[7] C. Asato *et al.* "A data-path multiplier with automatic insertion of pipeline stages," *IEEE J. Solid-State Circuits*, vol. 25, pp. 383-387, Apr. 1990.

[8] C. C. Stearns and P. H. Ang, "Yet another multiplier architecture," in *Proc. IEEE Custom Integrated Circuits Conf.* (Boston), May 1990, pp. 24.6.1-24.6.4.

[9] T. Sato *et al.*, "A regularly structured 54-bit modified Wallace tree multiplier," in *Proc. IFIP Int. Conf. VLSI (VLSI91)* (Edinburgh), Aug. 1991, pp. 1.1.1-1.1.9.

[10] *ANSI/IEEE Standard 754—1985 for Binary Floating—Point Arithmetic.* Los Alamitos, CA: IEEE Computer Soc., 1985.

[11] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, pp. 236-240, 1951.

[12] E. Hokenek *et al.*, "Second-generation RISC floating point with multiply-add fused," *IEEE J. Solid-State Circuits*, vol. 25, pp. 1207-1213, Oct. 1990.

[13] M. Nagamatsu *et al.*, "A 15-ns 32 × 32-b CMOS multiplier with an improved parallel structure," *IEEE J. Solid-State Circuits*, vol. 25, pp. 494-497, Apr. 1990.

[14] J. Mori *et al.*, "A 10-ns 54 × 54-b parallel structured full array multiplier with 0.5-μm CMOS technology," *IEEE J. Solid-State Circuits*, vol. 26, pp. 600-606, Apr. 1991.

[15] *The TTL Databook for Design Engineers*, 2nd ed., Texas Instruments Inc., 1976, pp. 7-385.

[16] T. Sato *et al.*, "An 8.5-ns 112-bit transmission gate adder with a conflict-free bypass circuit," in *Symp. VLSI Circuits, Dig. Tech. Papers* (Oiso, Japan), May 1991, pp. 105-106.

[17] R. Turn, *Computers in the 1980s.* New York: Columbia University Press, 1974.

[18] R. Sharma *et al.*, "A 6.75 ns single metal CMOS 16 × 16 multiplier IC," in *Symp. VLSI Circuits, Dig. Tech. Papers* (Tokyo), Aug. 1988, pp. 91-92.

[19] K. Yano *et al.*, "A 3.8-ns CMOS 16 × 16-b multiplier using complementary pass-transistor logic," *IEEE J. Solid-State Circuits*, vol. 25, pp. 388-395, Apr. 1990.

A 4.4 ns CMOS 54 × 54-b Multiplier Using Pass-Transistor Multiplexer

Norio Ohkubo, Makoto Suzuki, *Member, IEEE,* Toshinobu Shinbo, Toshiaki Yamanaka, *Member, IEEE,* Akihiro Shimizu, Katsuro Sasaki, *Member, IEEE,* and Yoshinobu Nakagome, *Member, IEEE*

***Abstract*— A 54 × 54-b multiplier using pass-transistor multiplexers has been fabricated by 0.25 μm CMOS technology. To enhance the speed performance, a new 4-2 compressor and a carry lookahead adder (CLA), both featuring pass-transistor multiplexers, have been developed. The new circuits have a speed advantage over conventional CMOS circuits because the number of critical-path gate stages is minimized due to the high logic functionality of pass-transistor multiplexers. The active size of the 54 × 54-b multiplier is 3.77 × 3.41 mm. The multiplication time is 4.4 ns at a 2.5-V power supply.**

I. Introduction

ENHANCING the performance of floating-point operation is indispensable for current high-performance microprocessors. In particular, high-speed multiplication is becoming increasingly important in RISC's, DSP's, graphics accelerators, and so on, because of increasing demand for multimedia applications. Recent high-end microprocessors call for an operating frequency of 200 MHz or over. Furthermore, a multiplier will be required for single-clock-cycle operation. However, no CMOS 54 × 54-b multiplier with a delay time less than 5 ns has yet been reported [1], [2].

This paper describes a 54 × 54-b multiplier macro developed formantissa multiplication of 2 double-precision numbers, as outlined in the IEEE standard [3]. The target multiplication time is less than 5 ns. To reduce the multiplication time, a new 4-2 compressor and a carry lookahead adder (CLA), both featuring pass-transistor multiplexers, have been developed. The new circuits provide a speed advantage over conventional CMOS circuits because the number of critical-path gate stages is minimized due to the high logic functionality of pass-transistor multiplexers. In addition, power reduction is important in attaining such high performance. For this purpose, we employed 0.25 μm CMOS technology and reduced the supply voltage to 2.5 V.

The architecture of the 54 × 54-b multiplier is described in Section II. In Section III, the circuit design based on Booth's algorithm, as well as the design of the pass-transistor multiplexer, 4-2 compressor, and carry lookahead adder are discussed. Section IV describes the fabrication of a test chip. Some experimental results are shown in Section V, and the conclusions are summarized in Section VI.

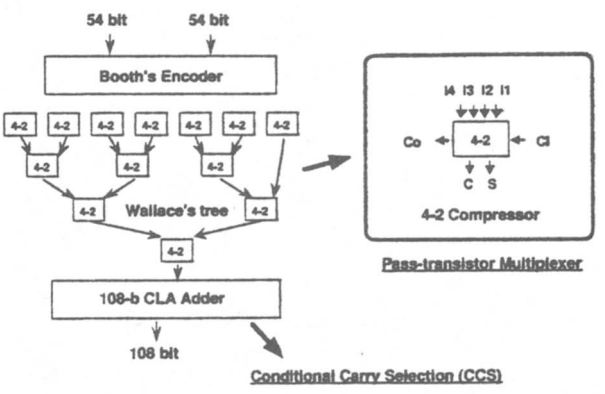


Fig. 1. Block diagram of the 54 × 54-b multiplier usingpass-transistor multiplexers.

II. Architecture

The block diagram of the 54 × 54-b multiplier is shown in Fig. 1. It employs Booth's algorithm [4], Wallace's tree [5], and a conditional carry-selection (CCS) adder [6]. The number of partial products is halved by Booth's algorithm. The partial products are summed by Wallace's tree, without carry propagation. The summed results are then added by the CCS adder with high-speed carry propagation.

Reducing the delay of Wallace's tree is important in reducing multiplication time, so we used a 4-2 compressor, which has 5 inputs and 3 outputs. The carry-out (Co) is connected to the next higher bit 4-2 compressor's carry-in (Ci), as shown in Fig. 1. Without propagating the carry to a higher bit, the 4-2 compressor can add four partial products (I1-I4) because the carry-out (Co) does not depend on the carry-in (Ci). By using this 4-2 compressor, only four addition stages are needed for Wallace's tree, as shown in Fig. 1. It is known that the 4-2 compressor has a speed advantage over full-adder-based designs, because of the reduced number of addition stages [1], [2]. For further improvement, we have developed a new 4-2 compressor that reduces the critical path gate stages by exploiting the high logic functionality of the pass-transistor multiplexer.

Manuscript received August 1, 1994; revised 10/21/94.

N. Ohkubo, M. Suzuki, T. Yamanaka, and Y. Nakagome are with the Central Research Laboratory, Hitachi, Ltd., Kokubunji, Tokyo 185, Japan.

T. Shinbo and A. Shimizu are with the Hitachi VLSI Engineering Corporation, Kodaira, Tokyo 187, Japan.

K. Sasaki is with the R&D Division, Hitachi America, Ltd., Brisbane, CA 94005 USA.

IEEE Log Number 9408745.

Reprinted from *IEEE Journal of Solid-State Circuits,* Vol. 30, No. 3, pp. 251-256, March 1995.

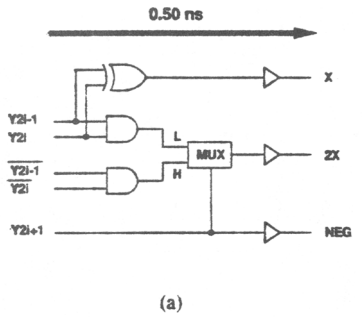


Fig. 2. Booth's algorithm. (a) Booth's encoder. (b) Partial-product generator.

Furthermore, we have developed a high-speed 108-b CLA adder, which is another important component of the high-speed multiplier. We have already reported a 32-b ALU with a 4-b lookahead carry scheme called conditional carry-selection [6]. To apply this scheme to the final adder of the multiplier, the 4-b CLA was modified into an 8-b CLA.

III. Circuit Design

A. Booth's Algorithm

The purpose of using Booth's algorithm in this design is not to reduce delay time, but to reduce the chip area. If the full-adder-based design is applied in Wallace's tree, the delay time can be shortened by using Booth's algorithm because it reduces the number of addition stages. However, because one addition stage of the 4-2 compressor halves the number of partial products, the extra delay time without Booth's algorithm is only that of one 4-2 compressor. Since the delay time in generating the partial products without Booth's encoder is shorter than that with Booth's encoder, the multiplication time, either with or without Booth's algorithm, are almost the same.

Booth's encoder is shown in Fig. 2(a). The multiplicands, $Y2_{i-1}$, $Y2_i$, and $Y2_{i+1}$, are encoded by this circuit. Encoding the data halves the number of partial products. The simulated propagation delay time is 0.50 ns. The partial-product generator is shown in Fig. 2(b). A multiplier, either X_j or X_{j-1}, is selected depending on whether encoded data, X or $2X$ is high and inverted by encoded data, NEG. The simulated propagation delay time is 0.56 ns.

B. Pass-Transistor Multiplexer

The pass-transistor multiplexer used in the 4-2 compressor and 108-b CLA adder is shown in Fig. 3. When the control signal S is low, data $D0$ is selected, and when the control

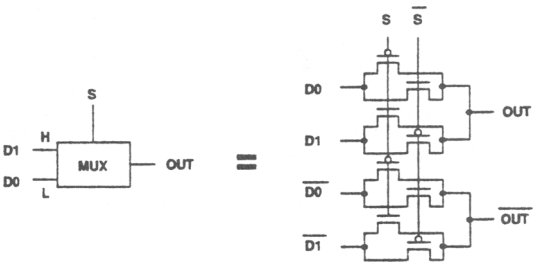


Fig. 3. Pass-transistor multiplexer circuit.

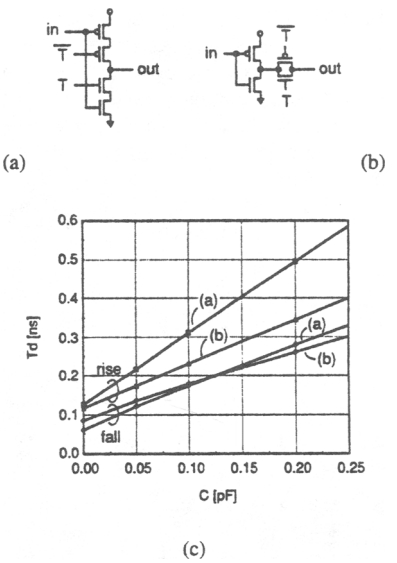
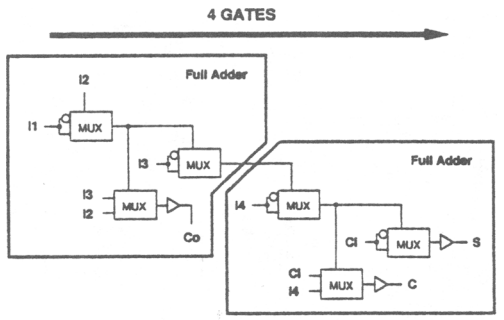


Fig. 4. Comparison of CMOS and pass-transistor logic circuits. (a) CMOS tristate inverter. (b) Pass-transistor tristate inverter. (c) Comparisonof delay time.

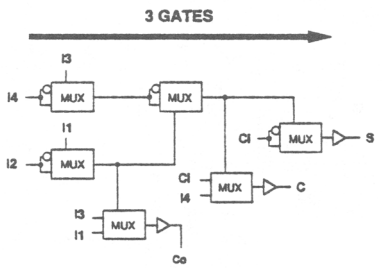
signal S is high, data $D1$ is selected. The output is used as the control signal input for the next-stage multiplexer. Thus, the multiplexer has both positive and negative output. It can reduce the propagation delay by eliminating an inverter.

Several pass-transistor logic circuits have been proposed to improve the performance of CMOS circuits. The NMOS pass-transistor logic circuits [7] is one example. It has been shown to result in high speed due to its low input capacitance and high logic functionality. However, particularly in reduced supply voltage designs, it is important to take into account the problems of noise margins and speed degradation. These are caused by mismatches between the input signal levels and the logic threshold voltage of the CMOS gates, which fluctuates with process variations. To avoid these problems, the multiplexer in this design consists of both NMOS and PMOS pass transistors.

The delay time of the pass-transistor multiplexer is shorter than that of a CMOS gate, because of the pass-transistor-based design where both the NMOS and PMOS are turned on. The CMOS tristate inverter and pass-transistor tristate inverter are shown in Fig. 4(a) and (b), which are used in CMOS and pass-transistor multiplexers, respectively. The number of transistors in both circuits is the same, and both have



(a)



(b)

Fig. 5. 4-2 compressor circuits using pass-transistor multiplexers. (a) Full-adder-based construction. (b) Proposed construction.

equal input capacitance. A simulated comparison is shown in Fig. 4(c), showing the dependence of the delay time from "in" to "out" on the output load capacitance. The low driving-source impedance attained by using the pass-transistor makes the delay time of the pass-transistor shorter than that of a CMOS gate.

C. 4-2 Compressor Circuit

The 4-2 compressor circuits using pass-transistor multiplexers are shown in Fig. 5. The signal lines in the figure represent positive and negative signals. The inputs of the multiplexers are either two different signals, or one signal and logical invert. All outputs (S, C, and Co) have buffers to enhance driving ability. The 4-2 compressor circuits add four partial products (I1–I4) and generate a sum signal (S) and two carry signals (C and Co).

Since the pass-transistor multiplexer circuit shown in Fig. 3 has high logic functionality, a full-adder circuit is constructed from three pass-transistor multiplexers. The 4-2 compressor is constructed from 2 full adders, such that there are 4 critical-path gate stages, as shown in Fig. 5(a). This circuit is faster than conventional CMOS circuits due to the use of pass-transistor multiplexers.

For further speed improvement, we developed a new 4-2 compressor. Though the number of multiplexers is the same, the number of critical-path gate stages in this circuit is reduced to 3 by exploiting parallelism, as shown in Fig. 5(b). In this new configuration, the carry-out (Co) does not depend on the carry-in (Ci), so, the advantage of the 4-2 compressor is maintained with this new configuration. The simulated delay

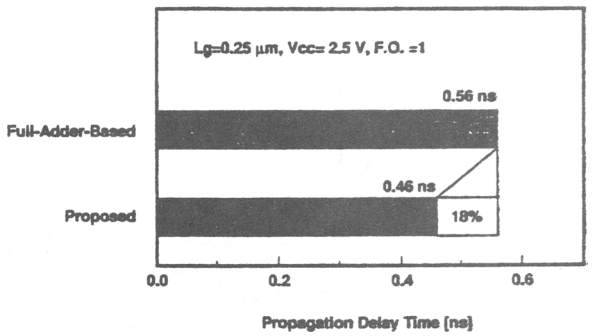


Fig. 6. Simulated comparison of 4-2 compressor circuits.

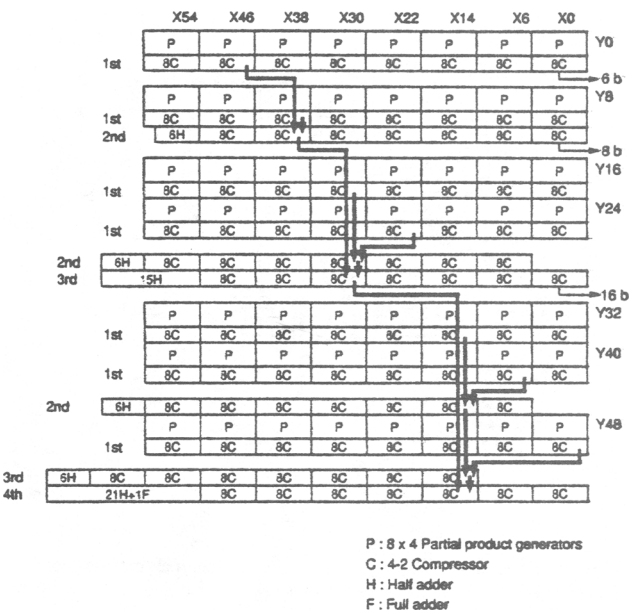


Fig. 7. Construction of Wallace's tree.

comparison for these 4-2 compressor circuits is shown in Fig. 6 The proposed circuit reduces the propagation delay time by 18% from that of a full-adder-based circuit.

Wallace's tree, shown in Fig. 7, is constructed from partial-product generators, 4-2 compressors, full adders, and half adders. Using the 4-2 compressor simplifies the construction of Wallace's tree. The Wallace's tree consist of only five kinds of blocks and four kinds of wire shifters. The five kinds of blocks include an 8 × 4 partial-product generator (P), eight 4-2 compressors ($8C$), six half adders ($6H$), 15 half adders ($15H$), and 21 half adders and a full adder ($21H + 1F$). The four kinds of wire shifters include an 8-b right shifter, an 8-b left shifter, a 16-b right shifter, and a 16-b left shifter. The sum and carry signals of 4-2 compressors are shifted by the wire shifters. In addition, the carry signals are shifted to the left one more bit.

D. Conditional Carry-Selection Circuit

A number of fast-adder architectures have been proposed [8]–[14], some of which use pass-transistor logic circuits for carry propagation [13], [14]. These circuits gain their

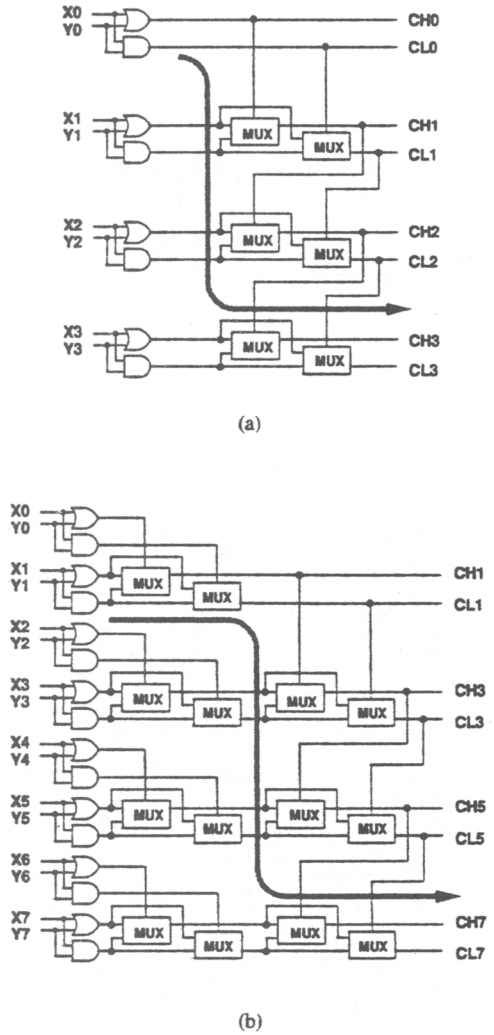


Fig. 8. Carry lookahead circuits. (a) 4-b conditional carry-select (CCS) circuit. (b) 8-b modified conditional carry-selectcircuit.

speed advantage over CMOS circuits due to their high logic functionality. However, a problem with this architecture is the serial connection of the pass transistors in the carry propagation path.

We have already reported a new look ahead carry scheme called conditional carry-selection (CCS) [6]. The 4-b CLA is constructed so as to have three critical-path gate stages, as shown in Fig. 8(a). In the CCS architecture, conditional carry signals for each bit, CL_j (assuming an incoming group-carry of "0") or CH_j (assuming an incoming group-carry of "1"), are selected by the multiplexers depending on the conditional carry signals of the previous bit, CL_{j-1} or CH_{j-1}, as expressed by

$$\begin{aligned} CL_j &= G_j + P_j \cdot C_{j-1} = G_j + P_j \cdot 0 \\ &= G_j = X_j \cdot Y_j \qquad (1) \\ CH_j &= G_j + P_j \cdot C_{j-1} = G_j + P_j \cdot 1 \\ &= G_j + P_j = X_j + Y_j \qquad (2) \end{aligned}$$

where G_j is carry generation and P_j is carry propagation. The CCS adder is faster than the conventional pass-transistor-based design because the pass-transistors are not serially connected

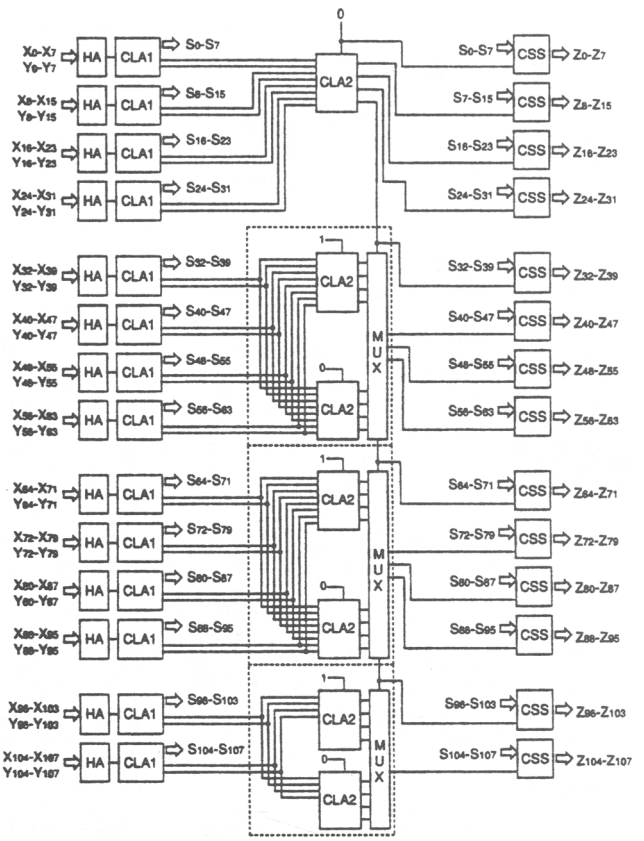


Fig. 9. Construction of 108-b adder.

in the carry propagation path. This is because the multiplexer's output is directly connected to the next multiplexer's control signal input.

To apply the CCS scheme to the final 108-b adder, the 4-b CLA is modified into an 8-b CLA, as shown in Fig. 8(b). In the modified CCS architecture, conditional carry signals for each bit, CL_j or CH_j, are selected by the multiplexers depending on the conditional carry signals of the two previous bits, CL_{j-2} or CH_{j-2}, as expressed by

$$\begin{aligned} CL_j &= G_j + P_j \cdot C_{j-1} = G_j + P_j \cdot (G_{j-1} + P_{j-1} \cdot 0) \\ &= G_j = X_j \cdot Y_j, \qquad \text{if } G_{j-1} = 0 \qquad (3) \\ &= G_j + P_j = X_j + Y_j, \qquad \text{if } G_{j-1} = 1 \qquad (4) \\ CH_j &= G_j + P_j \cdot C_{j-1} = G_j + P_j \cdot (G_{j-1} + P_{j-1} \cdot 1) \\ &= G_j = X_j \cdot Y_j, \qquad \text{if } G_{j-1} + P_{j-1} = 0 \qquad (5) \\ &= G_j + P_j = X_j + Y_j, \qquad \text{if } G_{j-1} + P_{j-1} = 0. \qquad (6) \end{aligned}$$

In this configuration, there are several series-connected pass-transistors, as shown in Fig. 8(b). However, the critical path of addition in the multiplier has no series-connected pass-transistors. This critical path is created because the $X0$ in the critical path of multiplier arrives later than the other bits. The new 8-b CLA has only four critical-path gate stages because it makes use of parallelism. It reduces the 108-b addition time to 1.52 ns from 1.82 ns.

Fig. 9 shows a construction of the 108-b adder based on carry-selected architecture [10]. DPL gates [6] are used for the

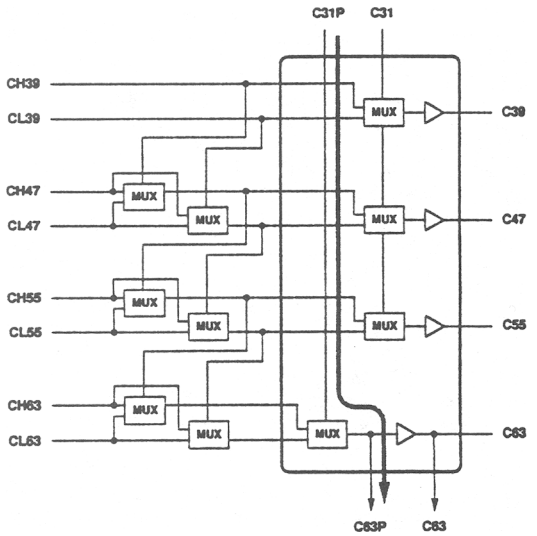


Fig. 10. Block carry lookahead circuit CLA2.

TABLE I
PROCESS TECHNOLOGY

Technology	0.25-μm CMOS Triple metal
Gate length	0.25 μm
Gate oxide	6.5 nm
1st Metal W/S	0.5 μm / 0.4 μm
2nd Metal W/S	0.5 μm / 0.4 μm
3rd Metal W/S	0.7 μm / 0.6 μm

TABLE II
CHARACTERISTICS OF THE 54 × 54-BIT MULTIPLIER

Organization	54 x 54-b
Delay	4.4 ns
Active Area	3.77 X 3.41 mm
Transistors	100,200
Supply Voltage	2.5 V

half adders (HA) circuits. The conditional-sum selection (CSS) circuit consists of a multiplexer which selects the conditional sums, $S_j(0)$ or $S_j(1)$, according to the incoming block carry signal. The CCS architecture is applied not only to the 8-b carry lookahead circuit CLA1, but also to the block carry look-ahead circuit CLA2, as shown in Fig. 10. The block carry signals are generated by CLA2 assuming the carry of the lower block carry signals to be 0 or 1, and are then selected by the multiplexer according to the incoming true carry. To shorten the carry propagation delay, the multiplexer of the critical carry propagation path is separated from the other multiplexers, as shown in Fig. 10. This architecture enhances parallelism and results in fast operation, because the carry signals of the upper 32 b are calculated in parallel with those of the lower 32 b, and the carry signals of the upper 32 b are generated after the delay time of a single multiplexer.

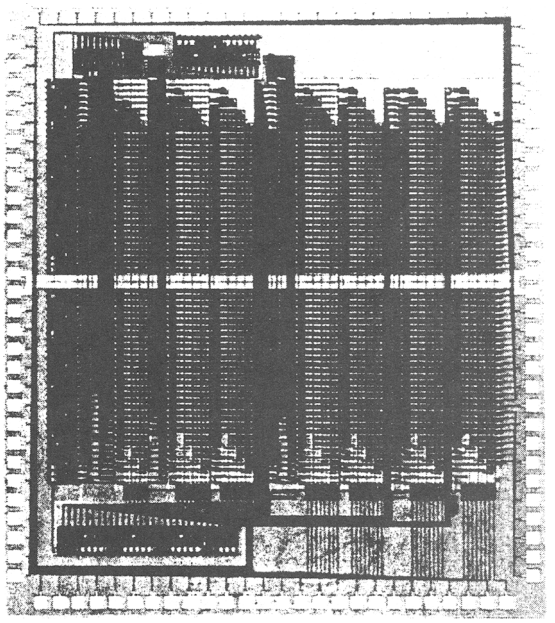


Fig. 11. Micrograph of the 54 × 54-b multiplier.

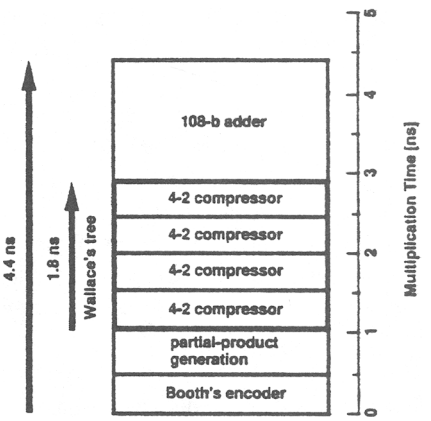


Fig. 12. Multiplication time of the 54 × 54-b multiplier.

IV. FABRICATION

A 54 × 54-b multiplier test chip was fabricated by triple-metal 0.25-μm CMOS technology. The major process parameters are summarized in Table I. The first metal is tungsten, and the second and third metals are aluminum. It operates on a supply voltage of 2.5 V. Fig. 11 shows a micrograph of the chip. The top and right-hand side are the input sections, Booth's encoder is also positioned on the right-hand side. The final adder and the output are at the bottom. 100 200 transistors are integrated in an active area of 3.77 × 3.41 mm. The area is mostly occupied by the 4-2 compressors, the partial-product generators, and wire shifters.

V. EVALUATION

The simulated multiplication time of the 54 × 54-b multiplier is shown in Fig. 12. It is only 4.4 ns with a 2.5-V power supply and typical process at room temperature. It

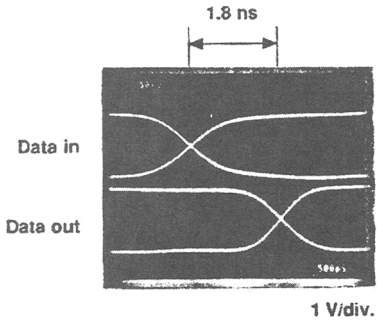


Fig. 13. Measured waveforms of Wallace's tree.

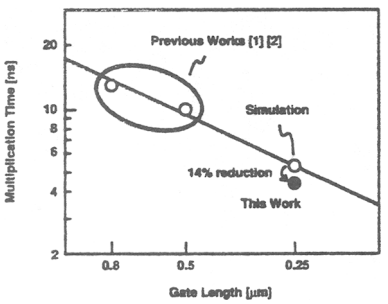


Fig. 14. 54 × 54-b multiplication time versus device dimensions.

shows excellent characteristics at such a low voltage because of the pass-transistor-multiplexer-based design where both the NMOS and PMOS are tuned on. The new 4-2 compressor reduces the Wallace's tree delay time to 1.8 ns. The characteristics of this 54 × 54-b multiplier test chip are summarized in Table II. The measured waveforms of Wallace's tree are shown in Fig. 13. The measured delay was almost the same as that of the simulated delay.

Fig. 14 shows the multiplication time plotted against the device dimensions. The multiplication time without the new circuits is estimated to be 5.1 ns, and the area of the multiplier does not increase significantly with the new circuits. Therefore, this multiplier achieves 14% improvement in multiplication time with the use of the new circuits having pass-transistor multiplexers.

VI. Conclusion

A new 4-2 compressor and CLA using pass-transistor multiplexers have been developed to shorten multiplication time. The multiplication time of the 54 × 54-b multiplier is reduced by 14% due to the reduction in the number of critical-path gate stages resulting from the use of pass-transistor multiplexers. A 4.4-ns multiplication time was achieved with 0.25-μm CMOS technology.

Acknowledgment

The authors wish to thank Dr. E. Takeda, Dr. K. Shimohigashi, Dr. T. Nishimukai, and Dr. T. Nagano for their useful discussions, and K. Ueda and K. Takasugi for their assistance and support. The authors are also greatly indebted to T. Nishida, N. Hashimoto, N. Ohki, and H. Ishida for their assistance with the process technology and device fabrication.

References

[1] G. Goto *et al.*, "A 54- × 54-bit regularly structured tree multiplier," *IEEE J. Solid-State Circuits*, vol. 27, pp. 1229–1236, Sept. 1992.
[2] J. Mori *et al.*, "A 10-ns 54- × 54-bit parallel structured full array multiplier with 0.5-μm CMOS technology," *IEEE J. Solid-State Circuits*, vol. 26, pp. 600–606, Apr. 1991.
[3] Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, IEEE Computer Society, Los Alamitos, CA, 1985.
[4] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, pp. 236–240, 1951.
[5] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 14–17, Feb. 1964.
[6] M. Suzuki *et al.*, "A 1.5-ns 32-b CMOS ALU in double pass-transistor logic," *IEEE J. Solid-State Circuits*, vol. 28, pp. 1145–1151, Nov. 1993.
[7] K. Yano *et al.*, "A 3.8-ns CMOS 16 × 16-bit multiplier using complementary pass-transistor logic," *IEEE J. Solid-State Circuits*, vol. 25, pp. 388–395, Apr. 1990.
[8] J. Sklansky, "An evaluation of several two-summand binary adders," *IRE Trans. Electron. Comput.*, vol. EC-9, pp. 213–226, June 1960.
[9] ——, "Conditional-sum addition logic," *IRE Trans. Electron. Comput.*, vol. EC-9, pp. 226–231, June 1960.
[10] O. J. Bedrij, "Carry-select adder," *IRE Trans. Electron. Comput.*, vol. EC-11, pp. 340–346, June 1962.
[11] C. L. Chen, "2.5-V bipolar/CMOS circuits for 0.25-μm BiCMOS technology," *IEEE J. Solid-State Circuits*, vol. 27, no. 4, pp. 485–491, Apr. 1992.
[12] K. Yano *et al.*, "3.3-V BiCMOS circuit techniques for 250-MHz RISC arithmetic modules," *IEEE J. Solid-State Circuits*, vol. 27, pp. 373–381, Mar. 1992.
[13] H. Hara *et al.*, "0.5-μm 3.3-V BiCMOS standard cells with 32-kilobyte cache and ten-port register file," *IEEE J. Solid-State Circuits*, vol. 27, pp. 1579–1584, Nov. 1992.
[14] A. Rothermel *et al.*, "Realization of transmission-gate conditional-sum (TGCS) adders with low latency time," *IEEE J. Solid-State Circuits*, vol. 24, pp. 558–561, June 1989.

A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach

VOJIN G. OKLOBDZIJA, FELLOW, IEEE, DAVID VILLEGER, AND SIMON S. LIU

Abstract—This paper presents a method and an algorithm for generation of a parallel multiplier, which is optimized for speed. This method is applicable to any multiplier size and adaptable to any technology for which speed parameters are known. Most importantly, it is easy to incorporate this method in silicon compilation or logic synthesis tools. The parallel multiplier produced by the proposed method outperforms other schemes used for comparison in our experiment. It uses the minimal number of cells in the partial product reduction tree. These findings are tested on design examples simulated in 1μ CMOS ASIC technology.

Index Terms—Parallel multiplier, partial product reduction, Wallace tree, Dadda's counter, VLSI arithmetic, Booth encoding, 3:2 counter, 4:2 adder, array multiplier.

1 Introduction

THE increased level of integration brought by modern VLSI and ULSI technology has rendered possible the integration of many components that were considered complex and were implemented only partially or not at all in the past. The multiplication operation is certainly present in many parts of a digital system or digital computer, most notably in signal processing, graphics and scientific computation. Therefore, it became quite common to see a multiplier implemented in full in many parts where it was not found before. Examples of such are floating-point processors and recently graphics processor, various kinds of digital signal processors used for user interfaces, communication or code compression. Parallel multipliers have even migrated into the fixed-point processor of digital computers for the purpose of speeding up and facilitating address calculation needed for fast and efficient indexing through arrays of data. The speed of the parallel multiplier has always been a critical issue and, therefore, the subject of many research projects and papers [1], [2].

Several popular and well-known schemes, with the objective of improving the speed of the parallel multiplier, have been developed in the past. The first departure from the iterative array structure [3] has been described in a paper by Wallace [7]. Wallace has introduced a notion of a carry-save tree constructed from one-bit Full Adders as a way of reducing the number of partial product bits in a fast and efficient way. The notion of counters and a generalization of the Wallace scheme have been described in the paper by Dadda [8] who also proposed a method that minimized the number of counters in a compression tree. A good survey of several possible schemes based on Dadda's method can be found in the paper by Stenzel [9].

In 1981, Weinberger [10] introduced a 4:2 compressor as a way of reducing the bits in the parallel multiplier array. His compressor was used by Santoro [12], Nagamatsu et al. [13], and Mori et al. [14]. The introduction of the 4:2 compressor (as an alternative to counters) was a departure from the traditional path which resulted in an improvement over the traditionally used Wallace and Dadda's scheme.

The use of larger compressors and families of compressors was explored by Song and DeMichelli [15]. They have also developed a 9:2 compressor and made a comparison with respect to their implementation and layout.

Use of higher order compressors yielded mixed results in some cases, however, it showed a general trend toward building compressors of larger sizes as a way of making incremental improvements in multiplier speed. In our research we took the approach of generating the compressors of maximal possible size, (i.e., the size of the multiplier) which yielded better results than the use of any previous compressor or counter family. Therefore, we gradually abandoned the notion of levels and undertook a design of an optimized one-level compressor which evolved into an optimization process involving the entire array.

Our method is based on the fact that not all inputs and outputs from a device used as a compressor (or counter) contribute equally to the delay. Therefore, we sort them in a way which favors the use of fast inputs and outputs in the paths that are critical to the speed while we assign slow inputs to the signal paths which belong to the domain where an increase in the delay is tolerable. In the creation process we examine the entire multiplier array and all the signals that are entering the compressor which lead to a

- *V.G. Oklobdzija is with Integration, Berkeley, California, and with the Department of Electrical and Computer Engineering, University of California, Davis, CA 95616. E-mail: vojin@ece.ucdavis.edu.*
- *D. Villeger is with Ecole Superieure d'Ingenieurs en Electrotechnique et Electronique, 93162 Noisy le Grand Cedex, France*
- *S.S. Liu is with Advanced Micro Devices, Sunnyvale, CA 94088-3453.*

Manuscript received Oct. 21, 1993; revised May 19, 1994.
For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number C96005.

Reprinted from *IEEE Transactions on Computers*, Vol. 45, No. 3, pp. 294-305, March 1996.

global rather than local optimization. This is characterized by the use of a particular compressor optimized for a minimal delay with respect to its inputs and outputs only. Our method also leads to a general solution for any chosen device (compressor or counter). By being used interactively in conjunction with improvement of a basic compressor device, this leads to an array which is optimized for speed down to the device level.

Although Booth recoding [4], [5] is widely used in parallel multipliers, it does not change the structure of the reduction tree. Moreover, its efficiency has been denied by several authors which was consistent with our findings [6], [11, [15]. As it has been shown in [6], one row of 4:2 compressors has the same effect as Booth encoding, which is achieved in less time. Given that the method presented here achieves this compression in even less time than the one using 4:2 compressors, the use of Booth encoding in parallel multipliers ceases to be advantageous for the range of technologies discussed in this paper.

We very recently became aware of an effort to build multipliers automatically via a program which attempts to compensate for the different speed of different paths in the multiplier by by-passing some stages [16]. While the approach taken is correct, which basically abandons the idea of counters or compressors, it does not take into account that different inputs (not only outputs) have different contribution to speed, as does our method. Unlike our method, half adders are used extensively in [16] which leads to worse results in terms of speed and cell number. Moreover, that method is less general and does not allow extension to device optimization. Similar approaches have also been applied in the process of "timing optimization" used in "logic synthesis," most notably in [28] and [29].

In the last section of this paper, we consider the final carry-propagate adder used to sum the partial products which have been reduced to two rows. The Final Adder delay is an addendum to the multiplier delay and, therefore, is critical. The approaches taken in designing the Final Adder were mainly concentrated on the raw speed of the adder and did not consider the specifics related to the uneven signal arrival profile of its inputs. Our selection of the Final Adder is based on these specifics. The structure of the Final Adder is *tuned* into the signal arrival profile so that the delay of the adder is minimized under these conditions which are specific to its application. We conclude that *tuning* of the Final Adder is more important than the its raw speed and that any application of a complex and hardware consuming scheme which is not optimized with respect to the signal arrival, would only constitute a waste of resources.

Finally, in the examples of various multipliers designed in 1.0μ CMOS ASIC technology, we show the advantages of our method and how it compares to the others.

2 Comparison of Different Partial Product Reduction Schemes

2.1 Wallace and Dadda Schemes

A major departure from iterative array realization of parallel multipliers has been introduced in papers by Wallace [7] and Dadda [8]. Common to both is the use of Carry-Save form in a compression tree in order to reduce the number of partial product bits to two rows. The advantage of trees is that their speed increases with the *log* of the operand length, while this increase is linear in the case of iterative arrays. Dadda introduced the notion of *counters* and a methodology of designing trees that are optimized in terms of cell number. Thus, Dadda has treated the bit reduction process as a number of *levels* (steps), the application of which results in the reduction of the number of rows of partial products. At every level the partial product bits are passed through devices designated as *counters*, the purpose of which has been the reduction of the number of rows after a pass through a level. A (p, q) counter is defined as a combinatorial network that determines the number of ones (active signals) among its p inputs producing a result on its q bit output in the form of a binary number (count). Essential to the counter is a process of summation of the input bits. The number of output bits must be sufficient to represent all the possible sums of n bits: $2^q - 1 \geq p$. A Full Adder can be treated as a (3, 2) counter, leading to a representation of the Wallace tree as a special case of Dadda. It would be ideal if it were possible to develop a higher order counter such as (7, 3) or (15, 4) with a speed that surpasses the speed of the same counters built by using Full Adders. There were several studies undertaken in the past, most notably by Stenzel [9], however, the use of a Full Adder as a (3, 2) counter is still the most prevailing.

The process applied by Wallace and Dadda can be summarized as follows: after generating the partial products, a set of counters reduces the partial product matrix but does not propagate the carries. The resulting matrix is composed of the sums and the carries of the counters. Another set of counters then reduces this new matrix and so on, until a two-row matrix is generated. Those two rows are summed up with a Final Adder to which we will refer to as a Carry Propagate Adder (CPA). This method takes advantage of the carry save form to avoid the carry propagation until the Final Adder. In this scheme the number of levels is crucial and will determine the speed of the multiplier.

2.2 Use of 4:2 and Higher Order Compressor

A major departure from the Wallace-Dadda scheme has been the introduction of a 4:2 compressor by Weinberger of IBM [10]. His idea was made visible by Santoro who used it in his PhD thesis [11] and by Mori et al. [14] who later implemented it. The 4:2 compressor, as shown in Fig. 3, made a major difference in the way cells are interconnected by introducing a *horizontal* path and, therefore, a limited propagation of the carry signal in the multiplier. Indeed 4:2 structure is not a counter, since two output bits cannot represent five possible sums of 4 bits. Thus, a carry out is necessary and subsequently a carry in. However, since the carry out is not dependent on the carry in, only a limited carry propagation occurs. There were several benefits from using a 4:2 compressor, such as simpler and more regular wiring of the multiplier tree. More notable is the introduction of the notion of a *horizontal* and a *vertical* signal path. Naturally application of 4:2 compressors led to faster realization of a multiplier. The existence of a *horizontal* path has

led to the idea of moving the *critical path* of the multiplier tree away from the center toward the most significant end where the depth of the column of partial product bits is smaller than in the middle [18]. The exploration of this idea has brought some mixed results, mainly because a major redesign of the compressors was necessary. However, it has shed enough light on the real problem of the reduction of the partial product bits.

Naturally, researchers tried to explore the idea of 4:2 compressors further and research in this direction, most notably by Song and DeMichelli [15] led to the introduction of the 6:2 and 9:2 family. The term family is being used because the new compressors introduced were simply built on 4:2 compressors and (3, 2) counters. For example, the 9:2 compressor consists of one 6:2 compressor and three (3, 2) counters, where in turn, the 6:2 compressor contains one 4:2 compressor and two (3, 2) counters. Nevertheless, a multiplier built using 9:2 compressors showed a speed advantage over one built using 4:2 compressors.

The speed comparison for three different multipliers using, respectively, (3, 2) counters, 4:2 and 9:2 compressors for the multiplier sizes ranging from 0 to 100 bits in equivalent XOR delays in the *critical* path is shown in Fig. 1. We redesigned the 4:2 and 9:2 compressors at the gate level in order to obtain the best possible performance. The difference in speed favors 9:2 over 4:2 and (3, 2). These results are in agreement with findings in [11], [13], and [15]. We use a normalized XOR delay as a measure of speed for a particular implementation of an algorithm or a method. There are several reasons for doing this:

1) the critical path in the multiplier consists of a path through a series of XOR gates independent the algorithm is used.
2) the speed comparison is made independent on the technology, which is characterized by its ability to realize a fast XOR function.

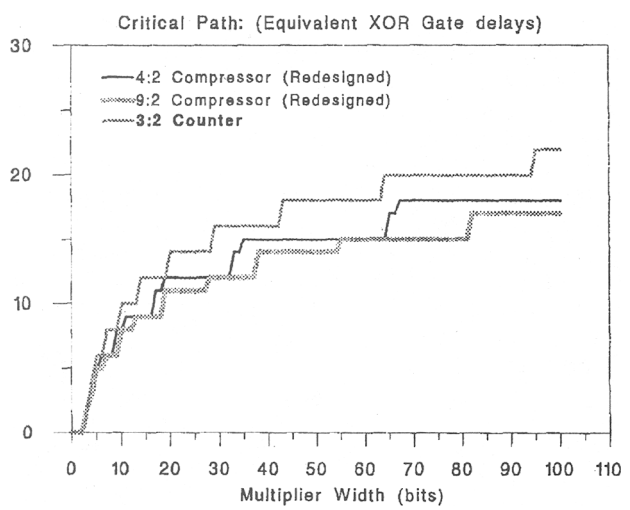


Fig. 1. Speed comparison for three different compressors (counters) used for partial product reduction: 9:2 is the best, followed by 4:2, and (3, 2).

In our early work, we followed this idea and came to the conclusion that the most efficient way to build a compressor is to start with the size of the multiplier (N), in other words to use the largest applicable size [18], [20]. The multipliers built in such a way resulted in better speed than those for which compressors of a smaller size were used [17], [18]. Though we could not claim that compressors built in such a way were optimal and would lead to the fastest realization, we identified the path to the solution. This path led us to the point where we were able to realize that the real problem is not in application or existence of a different counter or a compressor, but in the way the compressor tree has been interconnected. As we will see in the next section, any compressor can be designed with Full Adders with the speed of one designed at the gate level. Simply speaking, inside of each applied compressor, there is a Full Adder used as a building block. If that is the case, then a question arises to what is the difference? The answer is that the difference in using 4:2 and higher order compressors is not in the structure of the compressor but in the way they were interconnected.

2.3 Unequal Delays in a Full Adder: Existence of Fast Inputs and Fast Outputs

It is known that the delay from an input to an output in a Full Adder is not the same. This delay is even dependent on a particular transition (0-to-1, 1-to-0). It is also possible to come up with different realizations of a Full Adder where a specific signal path is favored with respect to the others and is designed in such a way that a signal propagation of this path takes a minimal amount of time. This is sometimes done even at the expense of other possible signal paths. For example, a ripple carry adder is designed so that the carry-in to carry-out delay is minimized. In that respect let us analyze a particular implementation of a (3, 2) counter as shown in Fig. 2. This particular case is taken from LSI 100K 1μ CMOS-ASIC cell [30]. It is used only for the illustration of the algorithm. In the case of a parallel multiplier, our design objective would be to minimize the delay from the *Input s* to the *Sum*, of the Full Adder which has direct effect on the *critical path*.

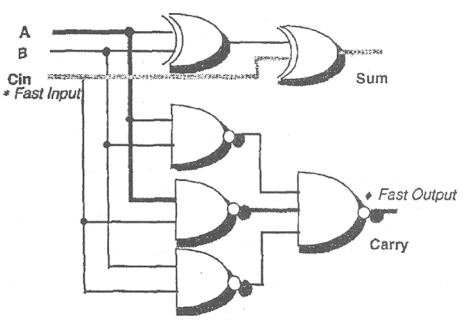


Fig.2. Signal delays in a Full Adder ((3, 2) counter).

In this example, the delay from input A or B to the Sum is equal to two equivalent XOR delays. The delay for the path from Cin to the output Sum is equal to one XOR

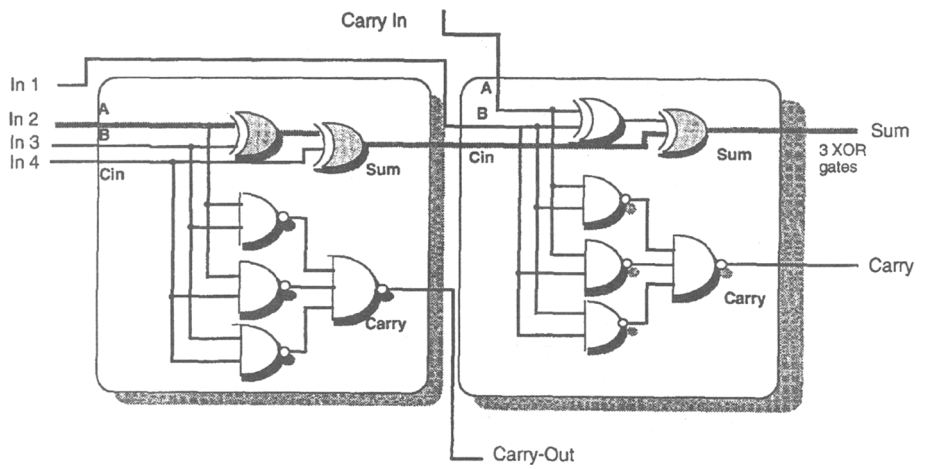


Fig. 3. Modified 4:2 compressor obtained by optimally interconnecting two Full Adders with fast input and output.

equivalent delay. We define Cin as a *fast input*. For this case, the propagation delay from A or B to the output Sum is twice as long as the propagation delay from input Cin to the Sum output. Considering the delay at the output Sum, in this particular technology delay from input A or B to Sum output is equivalent to two XOR delays. However, delay from inputs (A, B, or Cin) to the output Carry is equivalent to one XOR delay. We define Carry as a *fast output*. The value of those delays varies with technology and particular circuit implementation. In general we can use any (noninteger) values in our algorithm.

If we were to construct a 4:2 compressor by simply stacking two Full Adders together, as done by Santoro [11] the *critical path* of such a counter would be equal to four equivalent XOR delays. The researchers from Toshiba have simply redesigned the entire 4:2 compressor and treated it as a single cell. Their design resulted in three equivalent XOR gate delays and, therefore, they claimed 25% speed improvement over a conventional Full Adder Wallace tree realization.

In the next section, we will show how with proper interconnection the same speed can be achieved by using two regular Full Adders.

2.4 Improved 4:2 Compressor with Optimized Interconnections

The advantage of proper interconnection of *fast inputs* and *fast outputs* can be illustrated in the example of a 4:2 compressor. Fig. 3. shows an optimized 4:2 cell which is a result of applying our delay model and properly interconnecting *fast inputs* and *fast outputs* with the objective of minimizing the *critical path* of the 4:2 compressor. In our case, the delay through a 4:2 compressor level is equivalent to three XOR gate delays regardless of the path. If used to reduce the partial product bits in an 24 × 24-bit multiplier, the application of this modified 4:2 compressor would result in a delay of 12 equivalent XOR gates versus 16 if a regular 4:2 compressor, as used by Santoro, were applied (this numbers become 11 and 14, respectively, if a level of Full Adders is used every time the column size is three).

The use of the 4:2 compressor permits the reduction of the vertical critical path while the path involving the carry propagation, that we call *horizontal* path, is not changed. However, the *horizontal* propagation is fast and limited to 1 bit per level.

3 Three Dimensional Reduction Method

3.1 A New Approach

Instead of developing an efficient compressor structure and then using it in the process of partial product reduction, we took a global approach. In our method, we treat the entire multiplier array as a whole. The compressor consists of a vertical slice where the partial product bit array is represented in space and time (reduction steps). The vertical cross section in our representation (Fig. 4.), represents the partial product compressor of the multiplier array. The vertical slice in this representation is further interleaved with the compressors that are used in the reduction process and, therefore, represents a compressor structure corresponding to the appropriate bit position. We will refer to it as a Vertical Compressor Slice or VCS. It should be noted that there are a number of input signals into the vertical slice and a number of output signals originating from the particular vertical slice which are then being passed to the next VCS corresponding to the first higher order bit position.

Considering just one VCS we can see how the matrix of partial products is reduced by a tree of counters (Full Adders shown in the Fig. 4). However, every Full Adder produces a carry out which affects the slice of superior weight. Thus, the critical path is not only a *vertical* path through a given slice, but is also a *horizontal* path through the slices. As previously shown, the 4:2 compressor shortens the vertical path while including the horizontal path. The goal of the following scheme is to minimize both paths by building vertical slices that are optimized for a minimal delay. A method of designing VCS by considering both *vertical* and *horizontal* critical paths will be discussed.

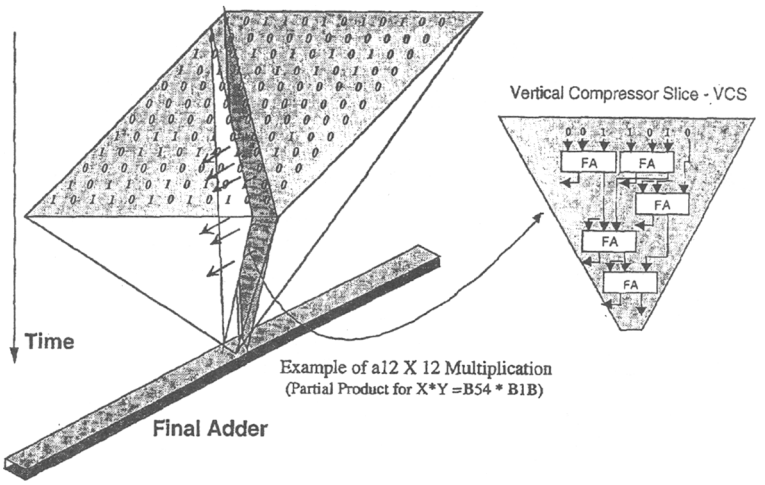


Fig. 4. A three-dimensional view of partial product reduction.

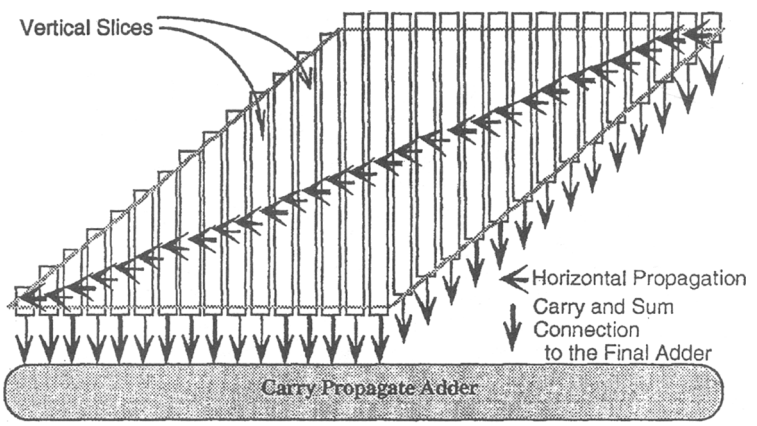


Fig. 5. The partial products matrix is divided into vertical compression slices.

The reduction of partial product bits in this new scheme is performed by creating a bit slice compressor VCS whose size is equal to the size of a given vertical cross-section of the partial product matrix, and then by assembling those compressors into an integral structure, as illustrated in Fig. 5. Since VCS are optimized by taking the neighboring VCS and their signals into account, as a contiguous process, the optimization is truly a three dimensional optimization process. We will refer to it as a Three Dimensional Minimization (TDM).

This approach is very different from Dadda's since all the partial products are compressed into a single step. This means that no intermediate partial products are considered in our case. However, TDM still produces a carry save number to be translated to a conventional form with a fast carry propagate adder (CPA). Indeed, each VCS reduces the partial products to a Sum and a Carry. Since it does not use the carry save form for the intermediate partial products, this scheme involves a carry propagation through the VCS. Therefore, it is necessary to design the VCS such that this propagation does not introduce a long delay. The main concern becomes the minimization of the *vertical* and *horizontal* critical paths rather than the number of Full Adder levels.

In addition, this scheme simplifies the design of the multiplier and its description in a hardware description language (VHDL). It also leads to an automatic generation of the partial product array by producing a net list of its signals and components. However, its efficiency will depend on the features of the VCS. The next section introduces a method of designing the VCS and subsequently the whole tree with Full Adders and half adders. This method, which automatically generates a net-list of the partial product array has been implemented in C language.

3.2 Method

The basic idea of this method is to make proper connections globally so that the delay throughout each path is approximately the same. The long delay path originating from the previous compressor should be connected to the short

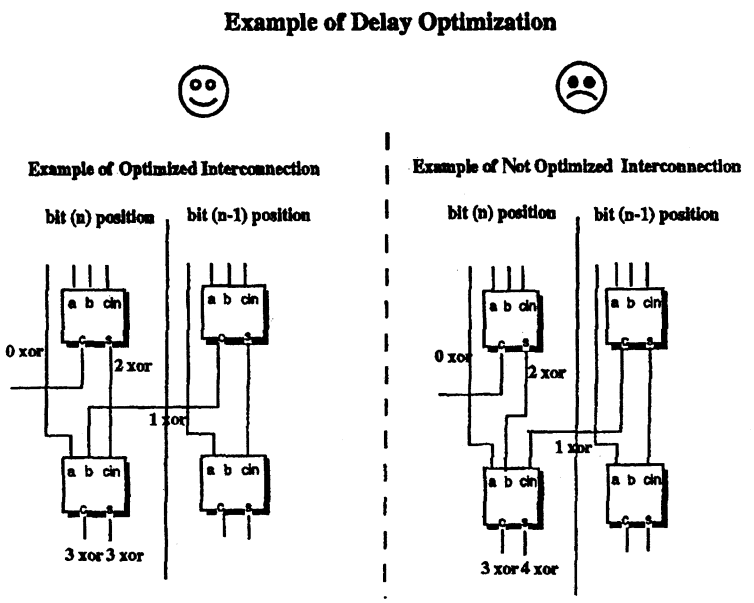


Fig. 6. Example of delay minimization by proper ordering of the input signals.

delay path of the next one, and so on. In general, this is not always possible since each output function has its unique characteristics and requires specific logic cells in its path. It is feasible to apply this idea to the partial product array using Full Adders (FA) since all of the partial product bits in the same bit position are logically the same and, therefore, interchangeable. In other words, all of the signals in any bit position can be interchanged no matter where they are originally coming from (as inputs in the same bit position, or as carries from a lower bit position). An example of how speed improvement can be achieved by application of this principle is shown in Fig. 6. The picture represents a small section of a multiplier tree. The application of this same principle resulted in the optimized 4:2 compressor shown in Fig. 3.

The presented method first creates a data structure consisting of 2N – 1 lists L_i containing names and delays of partial products. Each VCS is represented by a list of pairs $<d_j, n_j>_i$ containing delay and name information of a node. Initially, L_i represents the inputs (names and delays) of the corresponding VCS. Consequently, its length is the number of partial products from the corresponding slice and the delays are the delays produced by the partial product generator. Initially, we assume that all of the partial product bits are generated at the same time and we chose this to be a reference point by initializing all d_j to 0. If the partial products do not arrive at the same time, d_j will be assigned corresponding delay values. Some partial products do not need to be summed in the tree, and they are directly connected to the CPA.

After sorting the elements of L_i in ascending order by the values of the delays contained in the records of the list, a FA is connected to the first three nodes of L_i. The third node, that is the slowest one, is connected to the fast input (Carry-In) of the FA. The delays of the Sum and Carry are calculated and two new pairs $<d_j, n_j>_i$ are created containing information about those signals. The pair concerning the Sum signal is inserted in L_i while the one concerning the Carry signal is inserted in L_{i+1}. The size of L_i and L_{i+1} are then adjusted. This same procedure can be applied for any general type of (p, q) compressor cell used. The use of such a (p, q) compressor is advantageous only if such a cell shows speed advantages over FA in the particular technology of implementation.

The process stops when the size of L_i reaches three. The last three signals are then connected to a FA whose signals feed the CPA. Fig. 7 illustrates the effect of this method on different arrangements of signals that have different delays.

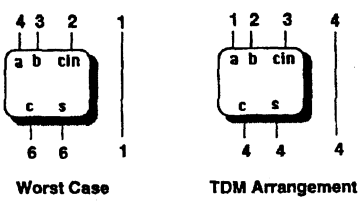


Fig. 7. Delay improvement with a different signal arrangement.

The delays of the Sum and Carry signal of the FA are calculated with the following equations:

$$\text{Delay(S)} = \text{MAX}\{\text{Delay(A)} + D_{A-S}, \text{Delay(B)} + D_{B-S}, \text{Delay}(C_{in}) + D_{Cin-S}\}$$

$$\text{Delay(C)} = \text{MAX}\{\text{Delay(A)} + D_{A-C}, \text{Delay(B)} + D_{B-C}, \text{Delay}(C_{in}) + D_{Cin-C}\}$$

where Delay(X) represents the delay attached to the signal X and the constants D_{U-V} represent the delays of a

signal crossing a FA from u to v. It should be noted that delays D_{u-v} can have any value. Those values are determined by the technology of implementation and circuit techniques used.

It is not always possible to use only FA in our design. In some cases, the use of a HA is necessary. The following demonstration shows that this use depends on the number of inputs of a VCS.

Let I_{CN} be the number of signals resulting from the application of *CN* counters. I_0 is the number of inputs of a VCS. Consequently, I_0 is the number of partial products at this particular bit position plus the number of carry signals originating from the previous VCS. It is not difficult to show that $I_{CN} = I_0 - CNR$ where R is defined as the difference between the number of inputs of the compressor and the number of Sum signals. In our case using FAs, R = 2 and $I_{CN} = 1$ (representing the Sum signal from the VCS). Then, $CN = (I_0 - 1)/2$. Since *CN* must be an integer, this expression is true if I_0 is odd. If I_0 is even, the number of Full Adders used to produce two signals is $CN = (I_0 - 2)/2$ which is an integer where the remaining two signals are reduced using a HA to produce a Sum and a Carry. Therefore, the number of cells can be expressed as:

$$CN_i = \lfloor I_i/2 \rfloor$$

In other words, a HA will be used if the number of inputs of a VCS is even. This HA is positioned near the partial product generator because carries that are generated at this position propagate through several slices. By taking advantage of the small carry delay generated by a HA, this positioning results in the gain of one XOR delay in the critical path of the multiplier.

The method, which generates the parallel multiplier bit compression array structure, is presented below. This method can be used as a basis for a program which generates a logic file containing the interconnection list as was done in our case. Such a program can be easily integrated into a silicon compiler or logic synthesis tool used for automatic generation of fast and efficient multiplier structure of any size.

3.3 Algorithm for Automatic Generation of Partial Product Array

Initialize:

Form 2N - 1 lists L_i (i = 0, 2N - 2) each consisting of p_i elements where:

$p_i = I + 1$ for $i \leq N - 1$ and $p_i = 2N - 1 - i$ for $i \geq N$

An element of a list L_i (j = 0, ..., p_{i-1}) is a pair: $<d_j, n_j>_i$ where:

nj : *is a unique node identifying name*

d_j : *is a delay associated with that node representing a delay of a signal arriving to the node n_j with respect to some reference point.*

For i = 0, 1 and 2N - 2: connect nodes from the corresponding lists L_i directly to the CPA;

```
For I = 2 to I = 2N - 3 {Partial Product
  Array Generation}
    Begin For
      if length of Li is even Then
        Begin If
          sort the elements of Li in
            ascending order by the values of
            delay dj;
          connect an HA to the first 2
            elements of Li starting with the
            slowest input;
              Ds = max {dA+dA-s, dB+dB-s}
              Dc = max {dA+dA-c, dB+dB-c}
          remove 2 elements from Li;
          insert the pair <Ds, NetName> into Li;
          insert the pair <Dc, NetName> into Li+1;
          decrement the length of Li;
          increment the length of Li+1;
        End If;
      while length of Li > 3
        Begin While
          sort the elements of Li in ascending
            order by the values of delay dj;
          connect an FA to the first 3 elements
            of Li starting with the slowest
            input of the FA:
              Ds = max {dcA+dcA-s, dcB+dcB-s,
                dcCi+dcCi-s};
              Dc = max {dcA+dcA-c, dcB+dcB-c,
                dcCi+dcCi-c};
          remove 3 elements from Li;
          insert the pair <Ds, NetName> into Li;
          insert the pair <Dc, NetName> into Li+1;
          subtract 2 from the length of Li;
          increment the length of Li+1;
        End While;
      sort the elements of Li;
      connect an FA to the last 3 nodes of Li;
      connect the S and C to the bit i and I + 1
        of the CPA;
    End For;
End Method;
```

The delay constants are technology dependent and are defined by the user.

3.4 Discussion of the Algorithm

The presented algorithm takes into account any delay value Ds and Dc as determined for the particular counter structure and technology of implementation. An average short wire delay and the effect of *loading* is included into Ds and Dc and calculated as a part of

$$Ds = \max \{dc_A + dc_{A-s}, dc_B + dc_{B-s}, dc_{Ci} + dc_{Ci-s}\}$$

and

$$Dc = \max \{dc_A + dc_{A-c}, dc_B + dc_{B-c}, dc_{Ci} + dc_{Ci-c}\}$$

expressions. It would be easy to generalize the same algorithm for use of higher order compressors (p, q). In such a

case, we calculate the delay D_i (i = 1, ... q) as:

$$D_i = \max \{d_j + d_{j-i} \text{ for } j = 1 \text{ to } p\}$$

Once the delays were calculated and list L_i has been sorted, we proceed by eliminating p elements from the list L_i and adding q delays to it. The objective of the algorithm is to minimize the delay of the signals that go to the next VCS. The critical path in the multiplier is usually found to be either in the largest VCS (in the middle of the multiplier tree), or as a path that starts at the least significant VCS and is being passed to the next until it ends up in the middle VCS. Therefore, our objective is to minimize:

1) the direct vertical path in a VCS (signals that are passed vertically to the FA),
2) the delay and the number of signals that are passed from VCS_i to VCS_{i+1}.

The problem of finding the multiplier structure with the minimal delay using arbitrary (p, q) counter is an NP hard problem, as is for finding a minimum delay wire layout. In the particular case presented in this paper, the problem is currently not known to be NP hard or polynomially solvable. Resolving this question is an interesting problem, and indeed we were able to prove that our algorithm produces an optimal and the best achievable structure as far as the speed of the multiplier is concerned [34].

The algorithm can easily be implemented to run in $O(n^2 \log n)$ time using priority queues, and in $O(n^2)$ time if the delays are small integers. We have not found a counter example which would yield better results for the cases of N < 64.

3.5 Example

In our case, the delays are :

$$FA_{A \to S} = FA_{B \to S} = 2 \text{ XOR delays}$$

$$FA_{Cin \to S} = FA_{A \to C} = FA_{B \to C} = FA_{Cin \to C} = 1 \text{ XOR delay.}$$

In the case of a HA, the delays become :

$$HA_{A \to S} = HA_{B \to S} = 1 \text{ XOR}$$
$$\text{delay while } HA_{A \to C} = HA_{B \to C} = 0.5 \text{ XOR delay.}$$

This is because our examples utilize LSI 100K: 1μ CMOS ASIC technology [30].

It is not difficult to generalize this method for the use of a general p:q compressor rather than a FA. This assumes that such a compressor can be built more efficiently than the one built from the FAs and that delay dependencies are known for the given technology of implementation.

An example representing the ninth VCS of a 12 × 12 multiplier is shown in Fig. 8. The numbers represent the delays that are associated to the nodes. The zero delays are the partial products and the nonzero delays at the top are carry signals originating from the previous slice. Since the number of inputs is even, the first cell used by the program is a HA. The reduction is achieved in five XOR delays.

In our example, we choose to normalize all the signal delays with respect to XOR gate delay and express delays as fractions of it. When different technology is used or the circuits are designed differently than in our case, the ratio of these delays can be quite different. For example, in [31] the sum delay is 1.5 times longer than the delay of the carry bit. However, even in such a case, re-connecting *fast-inputs* and *fast-outputs* according to the method presented here may result in the speed improvement of up to 18%.

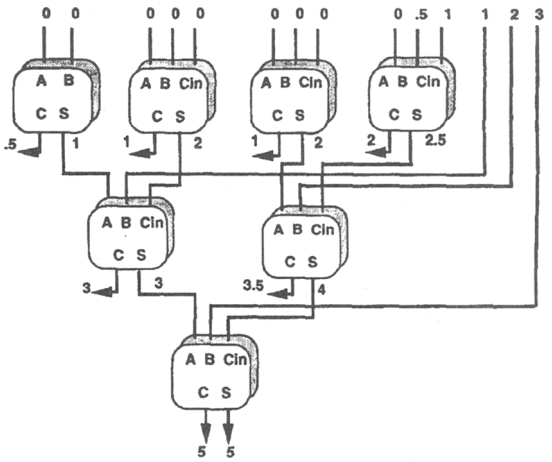


Fig. 8. The ninth VCS of a multiplier.

Although no Full Adder arrangement was assumed, the method produces a tree structure. This result was predictable since the tree structure is theoretically the fastest structure. As illustrated in the Fig. 8, the signals that have the longest delays will actually skip the next level of Full Adders. When it becomes impossible to skip more levels, they are connected to C_{in}, and when all C_{in} are occupied, they are connected to A or B. Since the delay of the carries are also calculated and used, the scheme minimizes both vertical and horizontal critical paths.

Table 1 compares the critical path in the partial product array produced by TDM and of the other schemes. The length of critical paths is estimated in the number of XOR levels. The delay of the *critical path* is dependent not only on the technology of implementation but also on the circuit family (dynamic or static) as well as layout and wiring delays. Therefore, it is difficult to give exact comparison, and the ones given in Table 1 are only reasonable estimates of the relative differences in delays with respect to other schemes [7], [11], [16]. We have decided to normalize all the delays to that of the XOR gate and use the XOR gate delay to represent respective delays in the multiplier tree structure. This decision is justified by the observation that the *critical path* in the multiplier tree indeed consists of a path through a series of XOR gates and that the ultimate speed of the multiplier (Final Adder included) indeed depends on how fast XOR gate can be implemented in a particular technology. A good example supporting this decision can be found in the pass-transistor multiplier implementation by Okhubo et al. [33].

This however, does not affect our method which is based on the *real delays* as calculated for the particular *counter* used in the multiplier under consideration.

TABLE 1
COMPARISON BETWEEN TDM AND OTHER REPRESENTATIVE SCHEMES, IN XOR LEVELS USED IN THE PARTIAL PRODUCT ARRAY

Multiplier Word-length	Wallace Tree [7]	4:2 Tree [11]*	Fadavi-Ardekani [16]	TDM
3	2	2	2	2
4	4	3	3	3
6	6	6 (5)	5	5
8	8	6	7	5
9	8	8	7	6
11	10	9 (8)	8	7
12	10	9 (8)	8	7
16	12	9	10	8
19	12	12 (11)	11	9
24	14	12 (11)	12	10
32	16	12	13	11
42	16	15 (14)	14	12
53	18	15	15	13
64	20	15	16	14
95	20	18 (17)	17	15

* *Number in parenthesis represent delays when a Full Adder is used (instead of 4:2 compressor) every time the column size is found to be three.*

Our algorithm and method can be extended further by going one level deeper into the hierarchy of design. We can treat the multiplier tree as a collection of *interconnected gates* or even *transistors* (as done in [33]) with designated *fast* and *slow* inputs and outputs. We leave this problem for the future extensions of this work suggesting this as a direction for possible further speed optimizations. However, assuming that we have optimized the signal paths using all the available circuit techniques in a particular counter, our method should yield the same results. Wiring delays were not given full consideration in our method.

The average wire delay is rather included into the gate delay as a function of fan-out. If the method presented here is to be fully integrated into the silicon compiler system, wire delay can be included and delays recalculated in an iterative process as each level of the bit reduction matrix is produced. In such a case it might be necessary to re-run the algorithm several time for possible corrections of the interconnection pattern and for fine-tuning of the delay list.

In the Wallace tree implementation, the number of XOR levels is simply the number of Full Adder levels multiplied by two. The improvement using TDM is up to 30%.

3.6 Hardware Complexity

In the following section, we show that the number of cells used is the same as the number produced by Dadda optimization.

The number of inputs to VCS_i is the sum of the number of partial products, called P_i, and the number of carries out from VCS_{i-1} which is actually $C_{i-1}-1$, since every cell of VCS_{i-1} produces a carry out except the one that is connected to the CPA. We have already shown that: $CN_i = \lfloor I_i/2 \rfloor$

THEOREM 1. *For $i \in [3, N-1]$ the number of cells in VCS_i, $CN_i = CN_{i-1} + 1$.*

PROOF.

The proof is by mathematical induction.

1) Initial step: It is true for I = 3. In this case, $CN_3 = 2$ and $CN_2 = 1$. Therefore, the theorem is true for the initial step
2) Induction hypothesis: $CN_i = CN_{i-1} + 1$.
3) Using the relation $CN_{i+1} = \lfloor (P_{i+1} + CN_i - 1/2 \rfloor$ and, since the induction hypothesis is $CN_i = CN_{i-1} + 1$ and $P_{i+1} = P_i + 1$ for $i \in [3, N-1]$, it follows that:

$$CN_{i+1} = \lfloor (P_i + CN_{i-1} - 1)/2 + 1 \rfloor = \lfloor (P_i + CN_{i-1} - 1)/2 \rfloor + 1.$$

Hence, $CN_{i+1} = CN_i + 1$ for $i \in [3, N-1]$.

THEOREM 2. *For $i \in [N, 2N-2]$ the number of cells in VCS_i, $CN_i = CN_{i-1} - 1$.*

PROOF.

1) Initial step: $CN_{N-1} = N - 2$ (application of the Theorem 1).
   For i = N,
   $CN_N = \lfloor (P_N + CN_{N-1} - 1)/2 \rfloor = \lfloor (2N-5)/2 \rfloor = N - 2.$
   Then, for i = N + 1
   $CN_{N+1} = \lfloor (P_{N+1} + CN_N - 1)/2 \rfloor = \lfloor (2N-6)/2 \rfloor = N - 3.$
   Therefore, the theorem is verified for the initial step
2) Induction hypothesis: $CN_i = CN_{i-1} - 1$.
3) Using the relation $CN_{i+1} = \lfloor (P_{i+1} + CN_i - 1) / 2 \rfloor$ and, since the induction hypothesis is $CN_i = CN_{i-1} - 1$ and $P_{i+1} = P_i - 1$ if $i \in [N, 2N-2]$, we find:

$$CN_{i+1} = \lfloor (P_i + CN_{i-1} - 1)/2 - 1 \rfloor = \lfloor (P_i + CN_{i-1} - 1)/2 \rfloor - 1.$$

Hence, $CN_{i+1} = CN_i - 1$ for $i \in [N, 2N-2]$.

THEOREM 3. *The total number of cells in the partial product array is $(N-1)(N-2)$*

PROOF.

VCS_0 and VCS_1 have no cells. The numbers of cells for VCS_2 to VCS_{N-1} represent an arithmetic progression $1 + 2 + 3 + \ldots + N - 2$. That is:

$$\sum_{i=1}^{N-2} i = \frac{(N-1)(N-2)}{2}.$$

The number of cells in VCS_N to VCS_{2N-2} is similarly shown to be $(N-1)(N-2)/2$ by using arithmetic progression $N - 2 + N - 1 + \ldots + 1$.

Therefore, the total number of cells is:

$$\sum_{i=0}^{2N-2} i = \sum_{i=0}^{N-1} i + \sum_{i=N}^{2N-2} i$$

$$= \frac{(N-1)(N-2)}{2} + \frac{(N-1)(N-2)}{2} = (N-1)(N-2).$$

The number of FA and HA produced by our program is exactly $(N-1)(N-2)$. Although Dadda did not give any formula [8], the results of his optimization are 110 cells for 12-bit multiplication and 506 cells for 24-bit multiplication. Those numbers correspond to our formula $(N-1)(N-2)$. Therefore, we claim that our method is also optimized in terms of number of cells. Fadavi-Ardekani (F-A) scheme produces 121 and 582 cells, respectively, for 12- and 24-bit multiplier sizes.

3.7 Comparison and Experimental Results

We have designed 4:2, 9:2, F-A, and TDM partial product arrays in 1μ CMOS-ASIC technology for a 24-bit multiplier. The results that were obtained by simulation using timing simulator from LSI Logic and LSI 100K [30] timing information under nominal conditions [T = 25°C, V_{cc} = 5V]. The results for the critical path delay in the partial product array are summarized in the Table 2. LSI Logic simulator takes into account loading due to different fan-outs and includes an estimated wire load. This estimation is based on the fan-out of the particular output node. Our past experience with the simulator based on comparisons with actual fabrications has been very good. However, our results are limited to simulation and they do not represent actual measurements. It is also appropriate to mention that our results do not reflect the complexity of wiring. The wiring required for a multiplier produced using the TDM method is not substantially different from other schemes such as Wallace or Dadda, knowing that the number of counters used by our method is comparable to Dadda's [8] and that the main difference is in the way inputs and outputs of the counters are connected locally, not in the additional connections. However, using an improved Dadda's scheme a multiplier can be designed with only local interconnections as done in [32]. TDM scheme uses more complex wiring than [32] and an *array multiplier* such as [31]. The use of (4, 2) compressors results in less complex wiring and layout.

TABLE 2
CRITICAL PATH DELAY [CMOS: LEFF = 1μ, T = 25°C, V_{CC} = 5V]

N = 24 bits	4:2 Design	9:2 Design	Fadavi-Ardekani	TDM Design
Delay [nS]	14.0	13.0	11.7	10.5

Fig. 9 depicts a comparison in terms of XOR levels between a regular Wallace/Dadda scheme, an optimized 4:2, the Fadavi-Ardekani scheme and the TDM scheme. The delay for the partial product array includes partial product generator delay consisting of a simple row of AND gates. In the next section, we consider the delay component of an optimal CPA.

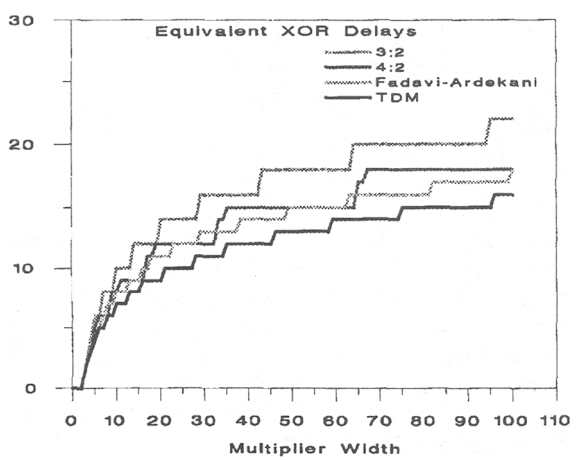


Fig. 9. Comparison of (3, 2), 4:2, F-A, and TDM schemes.

For this particular 24-bit CMOS implementation, the TDM Scheme is 33% faster than 4:2 and 25% and faster than 9:2. The improvement over F-A method is 11% in terms of delay and 15% in terms of the number of cells used.

4 Speed Improvement in the Final Adder

Finally, multiplier speed can be further improved via optimization of the CPA to the nonuniform signal arrival profile of its inputs. It is well-known that the signals applied to the inputs of the CPA arrive first at the ends of the CPA and the last ones are the signals fed to the bits in the middle of the CPA [26]. The shapes of the signal arrival profile originating from the CPA and the multiplier tree of an 13 × 13-bit ASIC multiplier [19] are shown in Fig. 10.

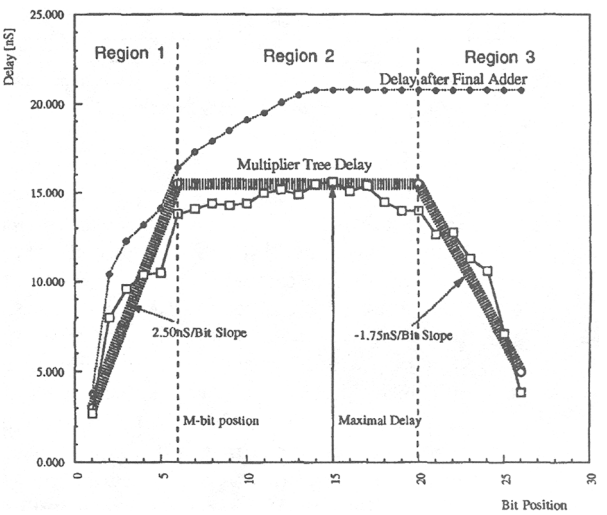


Fig. 10. Signal Arrival Profile and selection of the adder types in the three regions of the multiplier.

4.1 The Choice of Final Adder

All the known schemes for fast addition are developed under the assumption of an uniform signal arrival profile. The first problem is in selecting one of the CPA schemes that are most adequate to be used in the multiplier. It is obvious that we should use the fastest scheme since final addition time is a significant addendum to the critical path of the multiplier.

There are three regions to be considered with respect to the worst case signal arrival profile from the multiplier tree as shown in Fig. 10. Region 1 has a positive slope with respect to bit position. To use an adder which adds faster than this slope would not make much sense and would be a waste of hardware resources. The type of adder used in Region 1 is determined by this slope. This slope is determined by the fact that the arrival of bits is incrementally delayed by a path traversing a FA used in bit compression. Therefore, using any of the more powerful and, therefore, more complex, schemes for this part of the multiplier is not justified. If ripple-carry adder speed can not match this

slope, good choices are simple and VLSI efficient schemes such as Variable Block Adder (VBA) [23]. A simple analysis shows that CLA has worse performance in this region [27].

From the point of the maximal delay (M) which is usually in the middle bit position (or skewed a few bits toward the most significant side) the addition has to be performed in the fastest possible way. Therefore, in Regions 2 and 3, addition has to be very fast because the addition time Δ_{ADD_2} is a direct addendum to the multiplier delay. Analysis shows that the best choices for the adder in this region are: Conditional Sum Adder (CSA) and the Carry Select Adder (CSLA) [21], [27]. Choice of CSLA scheme results in less complex implementation simply because CSLA is a subset of CSA. The difference in speed between CSA and CSLA diminishes as the input arrival profile is changed from uniform to nonuniform [21]. Though the CSLA adder is still slower than the CSA adder, this difference is offset by the relative simplicity of its implementation which is preferred by most designers. Smaller size and simpler layout of CSLA would further affect the relative speed difference, reducing the advantage of CSA. Given that fact, Carry Select Addition (CSLA) is the best choice for Region 2 [22].

The adder constructed for this part (Regions 2 and 3) can be further subdivided into two. Region 2 requires the fastest addition available, the choice of which is Carry-Lookahead (CLA) adder, or optimized derivatives of CLA [24] combined into a CSLA, all of which depend on the particular circuit and technology used.

In the region of the negative slope (Region 3), where the most significant bits arrive first, the most suitable choice is again CSLA. However, CSLA adds time for selection multiplexers, ΔMUX, which is not negligible given that the adder sections are already constructed to be in the same speed range. It is also obvious that due to the steep negative slope in this region, there is substantial time left to add the bits in the most significant position before a selection process occurs. Simple analysis shows that for this part, VBA is a much better candidate than CLA or for that matter any other scheme [27].

Determination of bit positions separating the Regions 1, 2, and 3 is based on intersecting the bit arrival time from the adders used in the corresponding regions, so that the selection in CSLA is done at the appropriate time. This is a relatively complex iterative process which is illustrated in Fig. 11. As can be inferred from Fig. 11, this is an iterative technique which does not require a large number of iterations, given that the bit positions S_1 and S_2 are integers. The total delay of the multiplier is:

$$\Delta_{MULT} = \Delta_{TREE} + \Delta_{ADD_2} + \Delta_{MUX}$$

The resulting CPA structure is shown in Fig. 12a. The signal arrival profiles originating from the tree of an 13 × 13-bit multiplier for two different input patterns applied are shown in Fig. 12b. It is interesting to observe that while the pattern B results in a faster signal arrival from the multiplier array than pattern A, the product originating from the CPA for the pattern B has a longer delay than the pattern A. This example illustrates the point that it is more important to *tune* the CPA into the signal arrival profile than to apply the fastest available addition scheme. Further analysis of CPA optimization can be found in [27].

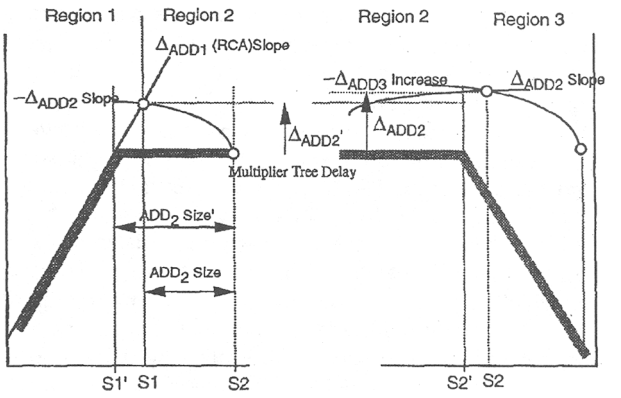


Fig. 11. Determination of bit positions S_1 and S_2 determining the size of the adders used: Left—Structure of the Final Adder; Right—Signal Arrival Profile.

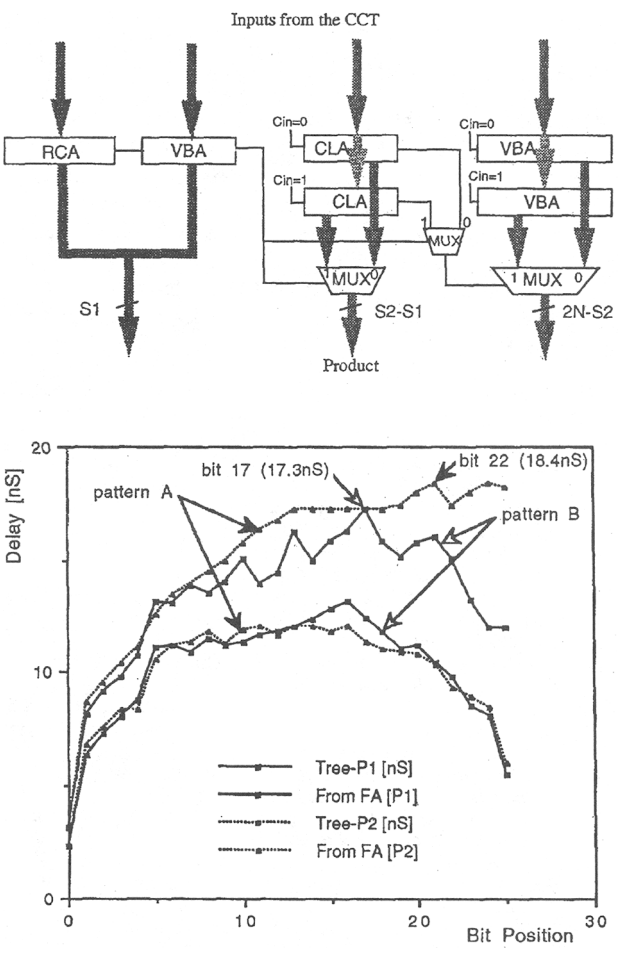


Fig. 12. Structure of the Final Adder (top); Signal Arrival Profile from the column compression tree and the Final Adder (bottom).

5 Conclusion

The presented algorithm and method for parallel multiplier implementation takes advantage of the uneven delays through a Full Adder in order to build a global compressor that minimizes the critical path of the multiplier. The compression tree is divided into vertical slices that are optimized globally to produce individual Vertical Compression Slices (VCS). This minimization does not only involve the vertical signal path, but also involves the horizontal signals from the previous VCS. A method to implement a speed optimized multiplier tree which includes an algorithm for net-list generation has been presented. This method has been implemented using C language, although it would not be difficult to describe in a hardware description language such as VHDL, or to implement the algorithm as a part of a silicon compiler. A multiplier produced by this algorithm has been implemented in 1μ CMOS technology together with several competing schemes. The results obtained using our algorithm are better not only in terms of speed but surprisingly also in terms of the number of cells used. The use of a compressor optimized at the transistor level instead of a Full Adder could result in further improvements of speed and can be easily incorporated in the presented method.

Acknowledgments

The authors are grateful to the reviewers. We are also grateful to Prof. Charles Martel for helping us understand the complexity of the algorithm. We express our gratitude to LSI Logic and Cascade Design Automation for supporting us generously with their tools, and to Antonio de la Serna for his careful reading of our text.

This research was supported by California Micro Grant No. 93-118 and by a grant from the Office of Research at the University of California at Davis.

References

[1] E.E. Swartzlander, *Computer Arithmetic*, vols. 1 and 2, IEEE CS Press, 1990.

[2] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley and Sons, 1979.

[3] S.D. Pezaris, "A 40ns 17-bit Array Multiplier," *IEEE Trans. Computers*, vol. 20, no. 4, pp. 442-447, Apr. 1971.

[4] A.D. Booth, "A Signed Binary Multiplication Technique," *Quarterly J. Mechanical Applications in Math.*, vol. 4, part 2, pp. 236-240, 1951.

[5] O.L. MacSorley, "High Speed Arithmetic in Binary Computers," *IRE Proc.*, vol. 49, pp. 67-91, Jan. 1961.

[6] D. Villeger and V.G. Oklobdzija, "Evaluation of Booth Encoding Techniques for Parallel Multiplier Implementation," *Electronics Letters*, vol. 29, no. 23, Nov. 1993.

[7] C.S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Trans. Computers*, vol. 13, no. 2, pp. 14-17, Feb. 1964.

[8] L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, vol. 34, pp. 349-356, Mar. 1965.

[9] W.J. Stenzel, "A Compact High Speed Parallel Multiplication Scheme," *IEEE Trans. Computers*, vol. 26, no. 2, pp. 948-957, Feb. 1977.

[10] A. Weinberger, "4:2 Carry-Save Adder Module," *IBM Technical Disclosure Bull.*, vol. 23, Jan. 1981.

[11] M.R. Santoro, "Design and Clocking of VLSI Multipliers," PhD dissertation, Technical Report No. CSL-TR-89-397, Oct. 1989.

[12] M.R. Santoro, "A Pipelined 64x64b Iterative Array Multiplier," *Digest of Technical Papers, Int'l Solid-State Circuits Conf.*, Feb. 1988.

[13] M. Nagamatsu et al., "A 15nS 32x32-bit CMOS Multiplier with an Improved Parallel Structure," *Digest of Technical Papers, IEEE Custom Integrated Circuits Conf.*, 1989.

[14] J. Mori et al., "A 10nS 54x54-b Parallel Structured Full Array Multiplier with 0.5-u CMOS Technology," *IEEE J. Solid State Circuits*, vol. 26, no. 4, Apr. 1991.

[15] P. Song and G. De Michelli, "Circuit and Architecture Trade-Offs for High Speed Multiplication," *IEEE J. Solid State Circuits*, vol. 26, no. 9, Sept. 1991.

[16] J. Fadavi-Ardekani, "M x N Booth Encoded Multiplier Generator Using Optimized Wallace Trees," *IEEE Trans. VLSI Systems*, vol. 1, no.2, June 1993.

[17] G. Bewick, "High Speed Multiplication," *Proc. Electronic Research Laboratory Seminar*, Stanford Univ., Mar. 12, 1993 (also private communications).

[18] V.G. Oklobdzija and D. Villeger, "Multiplier Design Utilizing Improved Column Compression Tree and Optimized Final Adder in CMOS Technology," *Proc. 10th Anniversary Symp. VLSI Circuits*, Taipei, Taiwan, May 1993.

[19] T. Soulas, D. Villeger, and V.G. Oklobdzija, "An ASIC Multiplier for Complex Numbers," *Proc. EURO-ASIC 93, The European Event in ASIC Design*, Paris, France, Feb. 22-25, 1993.

[20] D. Villeger, "Fast Parallel Multipliers," *Final Report, Ecole Superieure d'Ingenieurs en Electrotechnique et Electronique*, Noisy-le-Grand, France, May 11, 1993.

[21] A.K.W. Yeung and R.K. Yu, "A Self-Timed Multiplier with Optimized Final Adder," *Final Report for CS 292I* (Prof. Oklobdzija), Univ. of California at Berkeley, Fall 1989.

[22] O.J. Bedrij, "Carry-Select Adder," *IRE Trans. Electronic Computers*, June 1962.

[23] V.G. Oklobdzija and E.R. Barnes, "On Implementing Addition in VLSI Technology," *IEEE J. Parallel Processing and Distributed Computing*, no. 5, 1988.

[24] B.D. Lee and V.G. Oklobdzija, "Delay Optimization of Carry-Lookahead Adder Structure," *J. VLSI Signal Processing*, vol. 3, no. 4, Nov. 1991.

[25] M. Suzuki et al., "A 1.5nS 32b CMOS ALU in Double Pass-Transistor Logic," *Digest of Technical Papers, 1993 IEEE Solid-State Circuits Conf.*, San Francisco, Feb. 24-26, 1993.

[26] K. Fai-Pang et al., "Generation of High-Speed CMOS Multiplier-Accumulators," *Proc. ICCD-88, Int'l Conf. Computer Design*, Rye, N.Y., Oct. 1988.

[27] V.G. Oklobdzija, "Design and Analysis of fast Carry-Propagate Adder under Non-Equal Input Signal Arrival Profile," *Proc. 28th Asilomar Conf. Signals, Systems, and Computers*, Oct. 30-Nov. 2, 1994.

[28] K.J. Singh et al., "Timing Optimization of Combinational Logic," *Proc. ICCAD 88, Int'l Conf. CAD*, Nov. 1988.

[29] H.J. Touati et al., "Delay Optimization of Combinational Logic Circuits by Clustering," *Proc. ICCAD 91, Int'l Conf. CAD*, Oct. 1991.

[30] *1.0-Micron Array-Based Products Databook*, LSI Logic Corp., Sept. 1991.

[31] J.Y. Lee et al., "A High-Speed High-Density Silicon 8X8-bit Parallel Multiplier," *IEEE J. Solid State Circuits*, vol. 22, no. 1, Feb. 1987.

[32] Z. Wang, G.A. Julien, and W.C. Miller, "Column Compression Multipliers for Signal Processing Applications," *VLSI Signal Processing*. New York: IEEE Press, 1992.

[33] N. Okhubo et al., "A 4.4nS CMOS 54x54-b Multiplier Using Pass-Transistor Multiplexer," *Proc. Custom Integrated Circuits Conf.*, San Diego, Calif., May 1-4, 1994.

[34] C. Martel, V.G. Oklobdzija, R. Ravi, and P. Stelling, "Design Strategies for Optimal Multiplier Circuits," *Proc. 12th IEEE Symp. Computer Arithmetic*, Bath, England, July 19-21, 1995.

A 4.1-ns Compact 54 × 54-b Multiplier Utilizing Sign-Select Booth Encoders

Gensuke Goto, Atsuki Inoue, Ryoichi Ohe, Shoichiro Kashiwakura, Shin Mitarai, Takayuki Tsuru, and Tetsuo Izawa

Abstract—**A 54 × 54-b multiplier with only 60 K transistors has been fabricated by 0.25-μm CMOS technology. To reduce the total transistor count, we have developed two new approaches: sign-select Booth encoding and 48-transistor 4-2 compressor circuits both implemented with pass transistor logic. The sign-select Booth algorithm simplifies the Booth selector circuit and enables us to reduce the transistor count by 45% as compared with that of the conventional one. The new compressor reduces the count by 20% without speed degradation. By using these new circuits, the total transistor count of the multiplier is reduced by 24%. The active size of the 54 × 54-b multiplier is 1.04 × 1.27 mm and the multiplication time is 4.1 ns at a 2.5-V power supply.**

Index Terms— **Algorithm, CMOS digital integrated circuits, computer graphics, encoding, floating point numbers, IEEE standard, mulitplication, multiplying circuits.**

I. Introduction

BECAUSE of the rapid progress in very large scale integration (VLSI) design technologies, consecutive improvements in operational speed and design integration have been made. As a result of these improvements, interactive real-time three-dimensional (3-D) graphics applications have become available even on personal computers. In the geometric conversion processes of 3-D graphics applications, huge amounts of floating point operations are required and parallel data processing is inevitable. The amount of these arithmetic operations in real-time 3-D graphics applications depends on the quality of the 3-D objects and the frame rate. For example, in the case of a screen with a 1600 × 1280 resolution and at 60 frames/s, if 3-D objects are formed with small polygons (25 pixels/polygon) and 80% of the entire screen is constantly being processed with 3-D objects, approximately 4 Mp/s (million polygons per second) are processed. Our evaluations so far indicate that, to achieve 1 Mp/s of 3-D geometric conversion performance, 250 to 300 MFLOPS of floating point arithmetic performance are required.

Consequently, to achieve the above mentioned 4 Mp/s operation, more than 1 GFLOPS of floating point computation power is needed. In addition to 3-D graphics applications, a drastic improvement in the floating-point performance of general-purpose microprocessors and DSP's has been hoped for to meet the user demand for multimedia data processing. In such circumstances as above, a cost-effective, high-speed floating point computational core takes on a greater importance. In particular, the multiplier is one of the largest components of the datapath area, and a reduction of this multiplier core area is effective for this purpose.

This paper describes the method to reduce the transistor count and area of the multiplier core with the use of the new Booth encoding algorithm [1] and a new 4-2 compressor circuit [2], [3] for a carry-save adder (CSA) tree [4]. We applied this algorithm to a 54 × 54-b multiplier macro for mantissa multiplication of double-precision floating-point numbers, as outlined in the IEEE standard [5]. In our earlier design of a compact 54 × 54-b multiplier [6], nearly 90% of the transistor count was occupied with Booth selector circuits and the carry-save adders. We adopted Booth encoding using both negative and positive selection signals as encoded outputs. Thus, we call this method sign-select Booth encoding. With this encoder, the Booth selector circuit can be simplified and we succeeded in significantly reducing the transistor count in a multiplier. We have also developed new circuit implementation technology of 4-2 compressors that also enable us to reduce the transistor count in a carry-save adder tree.

The architecture of the 54 × 54-b multiplier and the signed Booth encoding algorithm is discussed in Section II. Section III describes the carry-save adder tree and circuit design based on a new 4-2 compressor. The fabrication of a test chip and some experimental results are described in Section IV. In Section V, a comparison with the other multiplier designs is made. Tne conclusions are summarized in Section VI.

II. Sign-Select Booth Encoding Algorithm

The block diagram of a 54 × 54-b tree-type multiplier is shown in Fig. 1. It employs the modified second-order Booth algorithm and a carry-save adder. The division scheme of Booth selectors (partial-product-bit generators) and adders in the multiplier is shown in Fig. 2. The carry-save adders are realized with the combination of 4-2 compressors 4 W and one-bit full adders 3 W. This is because 28 partial-product bits at maximum need to be compressed. Partitioning them into 7-b units composed of a 4- and a 3-W unit is useful for simplifying the complicated layout design. In the earlier 54 × 54-b multiplier design [6], carry-save adders occupy about 55% and the Booth selectors about 35% of the transistor count as shown in Fig. 1. This is because we need a large number of components for these two modules. In that design, 1527 Booth selectors and 625 4-2 compressors are used. However, the Booth encoder circuits consume 1.2%, as only 27 Booth encoder modules are necessary in that multiplier.

Manuscript received April 17, 1997; revised June 28, 1997.

G. Goto, A. Inoue, R. Ohe, and S. Kashiwakura are with Fujitsu Laboratories Ltd., System LSI Development Laboratories, Nakahara-ku, Kawasaki 211-88, Japan.

S. Mitarai, T. Tsuru, and T. Izawa are with Fujitsu Limited, System LSI Development Laboratories, Nakahara-ku, Kawasaki 211-88, Japan.

Publisher Item Identifier S 0018-9200(97)08032-3.

Reprinted from *IEEE Journal of Solid-State Circuits*, Vol. 32, No. 11, pp. 1676-1681, November 1997.

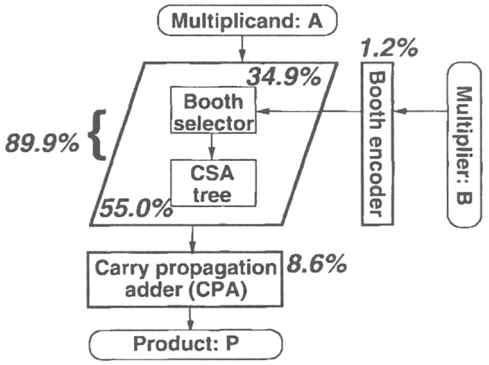


Fig. 1. Block diagram of multiplier. The italic numbers shows the percentage of number of transistors used in each part with respect to the total transistor count in our earlier multiplier design [6].

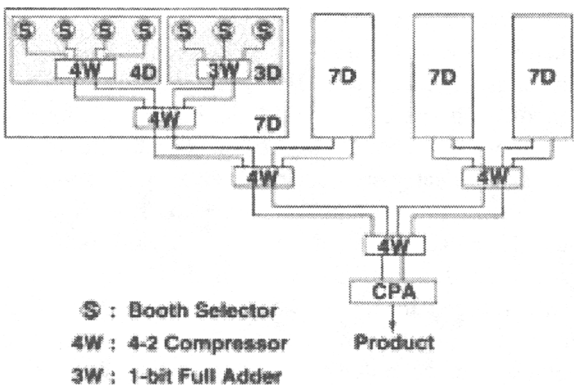


Fig. 2. Division scheme of Booth selectors and carry save adder tree in 54 × 54-b multiplier.

Thus, minimizing the volume of Booth selectors and carry-save adders is the key to establishing a compact multiplier.

The use of the Booth algorithm reduces the numbers of partial products and the carry-save adders significantly. In the second-order Booth encoding algorithm, we can reduce them by half. Consider the multiplication of two n-bit numbers in 2's complement, A and B, where we assume that n is an even number. The relative bit positions of the multiplicand and the multiplier are represented by a_i and b_j, respectively. The truth table of the usual second-order Booth encoder is summarized in Table I with that of the sign-select encoder. In the usual Booth encoder, three signals, X_j, $2X_j$, and M_j, are generated from the three adjacent multiplier bits, b_{j-1}, b_j, and b_{j+1} for selecting a partial product, that is, one of 0, $+A$, $-A$, $+2A$, and $-2A$. Here A is a multiplicand value of n bit width. The X_j and $2X_j$ signals show whether or not the partial product is doubled and M_j means the negative partial product. Thus, the multiplier value, either A or $2A$, is selected depending on whether the encoded data, X_j or $2X_j$, is high and the selected signal is inverted by encoded data, if M_j is the logical one. In this algorithm, a logical equation for the output signal, $P_{i,j}$, of the Booth selector at the ith multiplicand bit, a_i, and the jth multiplier bit, b_j, is given by

$$P_{i,j} = (a_i \cdot X_j + a_{i-1} \cdot 2X_j) \oplus M_j$$
$$(i = 0, 1, 2, \cdots, n-1 \quad j = 0, 2, 4, \cdots, n-4, n-2). \tag{1}$$

TABLE I
TRUTH TABLE FOR SECOND-ORDER MODIFIED BOOTH ENCODING

Inputs				Usual			Sign select			
b_{j+1}	b_j	b_{j-1}	Func.	X_j	$2X_j$	M_j	X_j	$2X_j$	PL_j	M_j
0	0	0	0	0	0	0	0	1	0	0
0	0	1	+A	1	0	0	1	0	1	0
0	1	0	+A	1	0	0	1	0	1	0
0	1	1	+2A	0	1	0	0	1	1	0
1	0	0	-2A	0	1	1	0	1	0	1
1	0	1	-A	1	0	1	1	0	0	1
1	1	0	-A	1	0	1	1	0	0	1
1	1	1	0	0	0	1	0	1	0	0

(j = 0,2,4,...,n-4,n-2)

P = A x B

$A = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$

$B = -b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j$

PL_j : Positive partial product
M_j : Negative partial product
X_j : Not doubled
$2X_j$: Doubled

Thus, the equation includes an exclusive-or (XOR) circuit to determine whether the partial product is inverted or not. A Booth encoder and a Booth selector require 32 and 18 transistors, respectively, for static CMOS logic implementation.

We developed a new Booth encoding algorithm with four output signals. We added an extra control signal PL_j to represent the selection of positive partial product. We introduced a sign-select Booth encoder with four output signals as shown in Fig. 3(a). In the usual encoder, only M_j, which represents the negative partial product, is generated. Conversely, in our sign-select encoder, signals for both the negative and positive partial products are generated. PL_j and M_j become active when the partial product is positive and negative, respectively. The X_j and $2X_j$ signals show whether or not the partial product is doubled. Using these encoded signals, we simplified the partial product generation logic. For example, when $+2A$ is necessary for the partial product, the PL_j and $2X_j$ signals are active, and the logical product of PL_j and $2X_j$ chooses $+2A$ as the partial product. When $-A$ is necessary, the logical product of M_j and X_j chooses the correct partial product, and so on. The logical equation for $P_{i,j}$ in this case is given by

$$P_{i,j} =$$
$$(a_i \cdot PL_j + \overline{a_i} \cdot M_j) \cdot X_j + (a_{i-1} \cdot PL_j + \overline{a_{i-1}} \cdot M_j) \cdot 2X_j$$
$$(i = 0, 1, 2, \cdots, n-1 \quad j = 0, 2, 4, \cdots, n-4, n-2). \tag{2}$$

From this equation, you will find that when both PL_j and M_j are inactive, "0" multiple as $P_{i,j}$ can be derived.

Thus, in our new Booth selector, exclusive OR logic is no longer necessary. Also, by utilizing pass transistor selectors, the Booth selector circuit is formed with fewer transistors. The circuit diagram is shown in Fig. 3(b) and (c) for both our new Booth selector and the conventional one. The selector (SEL) in Fig. 3(b) is implemented using pass transistor and consists of four transistors. The transistor count of the Booth selectors can be reduced from 18 transistors/b to 10 transistors/b when we combine two Booth selectors at consecutive multiplicand bit positions into a single module and sharing two inverters

TABLE II
SIMULATED CIRCUIT PERFORMANCE

Circuit	Transistor count in 4D	Delay time (ns)			
		Bit buffer +Booth encoder	Booth selector	4-stage 4-2 compressors	Total delay time
(1) OPPG+OD4W	132	0.34	0.40	1.99	2.73
(2) OPPG+O6T4W	116	0.34	0.41	3.05	3.80
(3) OPPG+DPL4W	124	0.31	0.46	2.05	2.82
(4) NPPG+MY4W	88	0.32	0.39	2.10	2.81

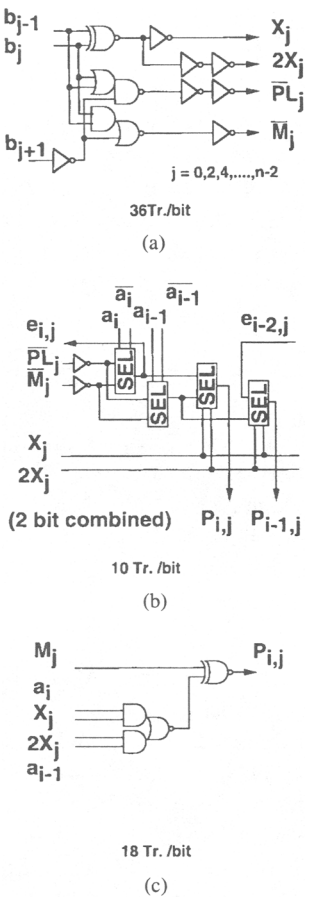


Fig. 3. Circuit diagrams for (a) Booth encoder, (b) new Booth selector used in this work, and (c) the conventional Booth selector.

through which we provide the selector with PL_j and M_j signals. By selecting these two signals by the multiplicand bit signal, a_i, sharing the inverters becomes possible with the selectors at the same jth multiplier bit position. In Fig. 3(b), the inverters are shared with two Booth selectors at consecutive multiplicand bit positions. Sharing the two inverters with more than two Booth selectors at consecutive multiplicand bit positions is logically possible, but at the cost of slower speed due to the larger RC time constant. The $\overline{a_i}$ and a_i signals used in the Booth selectors are buffered globally and the buffer circuits do not affect the total transistor count significantly. Although four more transistors are needed for each bit in the Booth encoder, eight transistors are eliminated from each bit in the Booth selector. As described before, the transistor count of Booth selectors occupies 35% of the space in a 54

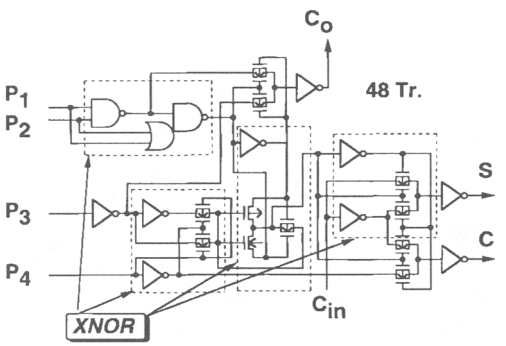


Fig. 4. Circuit diagram for 48-transistor 4-2 compressor.

× 54-b multiplier compared to only 1.2% for Booth encoders. The reduction in the number of transistors in Booth selectors thus has a greater impact on the total transistor count. The sign-select Booth encoding method enables us to reduce the transistor count by about 44% in our Booth selectors.

III. CARRY-SAVE ADDER TREE

The carry-save adder tree is the largest among the multiplier components. Therefore, we have to reduce the transistor count for the 4-2 compressors which are the constituents along with a one-bit full adder. The reduction is not easy, however, because a 4-2 compressor with fewer transistors usually needs a larger delay than that with more transistors. Careful optimization technology is required for the reduction of the transistor count for a 4-2 compressor without speed degradation.

There are four XOR circuits in a 4-2 compressor, and they are the major components in it. To implement an XOR circuit with fewer transistors and to share some transistors with an intermediate carry-generating circuit is key to overcoming these difficulties. There is an XOR circuit that is composed of six transistors [7]. A series-connected six-transistor XOR circuit is too slow to be applicable to a high-speed multiplier. However, we found that it is not too slow when sandwiched with other ten-transistor XOR circuits. We then adopted it in the middle part of the three-stage XOR chain, as shown in Fig. 4. In one of the first stages, we adopted an XOR circuit composed of a NAND gate and an OR–AND–inverter gate to share the NAND gate with an intermediate carry (C_o) generating circuit. By this sharing, an extra inverter is saved in the carry generator circuit. The other two XOR circuits are of a ten-transistor type composed of pass transistors and inverters. With this scheme

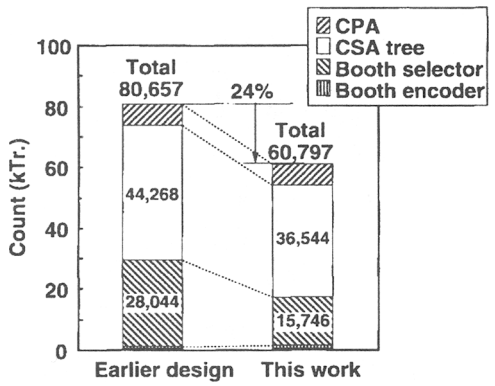


Fig. 5. Total transistor count compared with earlier design multiplier.

TABLE III
PROCESS TECHNOLOGY

Technology	0.25 μm CMOS 3-metal layer Co-Salicide
Gate Length	0.25 μm
Gate Oxide	5.5 nm
1st Metal (Line/Space)	0.44 μm / 0.46 μm
2nd Metal (Line/Space)	0.44 μm / 0.46 μm
3rd Metal (Line/Space)	0.44 μm / 0.46 μm
Planarization	CMP
Supply voltage	2.5 V

we reduced the transistor count of the 4-2 compressor from 60 to 48: a 20% reduction.

Table II shows the result of analysis using an HSPICE circuit simulator, and it compares our new circuits with the existing ones with respect to delay time of the 54 × 54-b multiplier except the final carry-propagate adder. The analysis is based on one process condition in 0.25-μm CMOS technology and all the n-channel or p-channel transistors are assumed to have the same width except for bit buffers. The wire lengths between the same kinds of blocks are assumed to be equal to one another among the models, and are in the same order as those of the test chip described in Section IV. OPPG and NPPG denote the Booth selectors that are implemented according to logical equations (1) and (2), respectively. OD4W, O6T4W, DPL4W, and MY4W are 4-2 compressors that are composed of the earlier design [6], all six-transistor XOR's [7], pass-transistor multiplexer logic [8], and the new 48-transistor logic, respectively. A 4D block consists of four Booth selectors and a 4-2 compressor 4W and generates and compresses four encoded bits at the same logical bit position. It is one of a basic block which is repeatedly arranged in a matrix to constitute a core part of the 54 × 54-b multiplier. In the table, the circuit (1), the combination of OPPG and OD4W is fastest, but the transistor count is also the highest. The difference in delay from circuits (3) and (4), however, is within 5% and it may be irrelevant as the accuracy of modeling for critical delay time is scattered among the circuits. Then we found that the three circuits (1), (3), and (4) have almost the same speed, and that circuit (2) is far slower than the other circuits, although the transistor count in a 4D block is lower than in circuits (1) and (3). Note that circuit (4), a combination of the new circuits, a Booth selector, and a 4-2 compressor, can constitute a 4D unit with fewer transistors than the alternative circuits and without any speed degradation as compared with the other competitors.

The total transistor count for our new 54 × 54-b multiplier is shown in Fig. 5, as compared with that of the earlier design of the regularly structured tree (RST) multiplier. By adopting the RST, we simplified the complicated carry-save adder tree wiring scheme and achieved a very compact design in less design time. We also obtained an area-efficient block because of the repetitive structure of the wiring, and thus the dedicated wiring area was minimized. For the final carry-propagate adder (CPA), we used the pass-transistor adder based on the combination of carry-skip and modified carry-select schemes that are used previously [9]. This adder can give sufficient speed for a performance-oriented multiplier with fewer transistors than other carry-propagate adders.

About a 24% reduction in the transistor count compared with the earlier design has been achieved for the whole multiplier. A total of 60797 transistors in a 54 × 54-b multiplier is the minimum count reported to date [6], [8], [10]–[13]. When we apply this method to a 26 × 26-b multiplier used for the mantissa multiplication of two single-precision numbers, we again obtain a 23% reduction in the transistor count, which means the logic we used is effective not only for larger multipliers [14], but also for smaller multipliers to reduce the transistor count.

IV. FABRICATION

We fabricated this multiplier using 0.25-μm CMOS technology. The major process parameters are summarized in Table III. The thickness of the gate oxide is 5.5 nm and salicidation process of $CoSi_2$ is utilized to form the source/drain area to reduce parasitic resistance. We used a triple-metal layer process with the CMP planarization technique. A photomicrograph of the chip is shown in Fig. 6. The chip integrates 60797 transistors in an active area of only 1.27 mm^2 except for the testing circuits. Trends regarding the chip area of a 54 × 54-b multiplier are summarized in Fig. 7. We can see that this is the minimum reported area for a 54 × 54-b multiplier, and we estimated a 21% reduction in the area as compared with the earlier RST multiplier made using the same fabrication technology. When we compare the area with the previously reported multiplier [10] with equivalent technology, it is ten times larger than ours. This is caused not only by larger transistor count but also by larger transistor size. Transistor count of the previously reported multiplier is 100200 which is 1.6 times larger than this work. The area of one transistor can be calculated as 128 μm^2/Tr. for the previously reported and 21 μm^2/Tr. for this work. Then the transistor size is six times larger than that of this work. The designed chip area reduction is directly reflected by the reduction in transistor count. Using the RST structure with the triple-metal process, we can minimize the wiring overhead in the Booth selector and CSA tree area. The measured wiring overhead by power

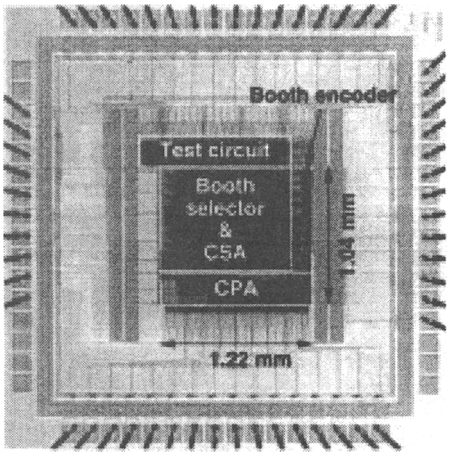


Fig. 6. A photomicrograph of the 54 × 54-b multiplier.

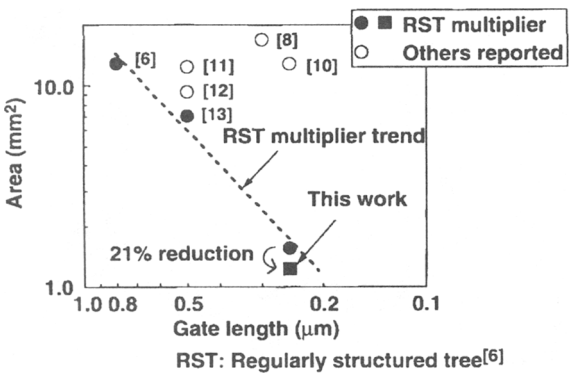


Fig. 7. Chip area trend for the 54 × 54-b multiplier.

lines and other wirings in this area is only 6.7%, thus the cell area dominates the entire array area in our RST layout.

V. Evaluation

Fig. 8 shows the measured waveform along the critical path using electron beam probing. A critical path delay of 4.1 ns was obtained at room temperature with a 2.5-V supply. This is a little faster than those of the multipliers fabricated with the equivalent technology [8], as summarized in Fig. 9. The measured delay time is 15% smaller than the simulated value in Section III, probably because of the parameter variation. By this measurement, we confirmed that our success in reducing the logic size does not diminish the speed performance. Since the simulated delay time of this circuit was almost the same as with others, the speed advantage is thought to be the smaller area of the present design compared to others as a result of reducing the transistor count, and also to the exclusion of wiring with the adoption of the RST structure. The previous design [10] necessitates a ten-times larger area than ours under a similar process condition, and this may require nearly a three-times larger wiring capacitance than that assumed in the model in Table II. The simulated delay time for all the circuits in the table is increased by 15% if the wiring capacitance between all the circuit units is doubled.

Taking these results into consideration, the data (OPPG + DPL4W) in Table II matches the previously reported data [8]. Since the earlier design of the RST multiplier in itself has an

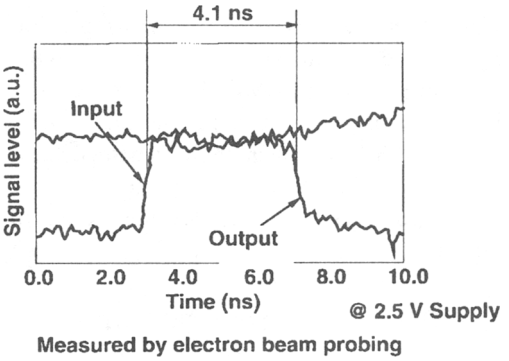


Fig. 8. Signal waveform for critical path.

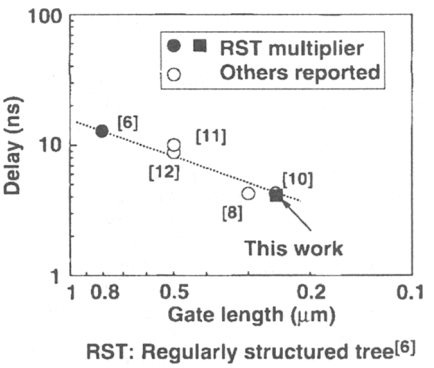


Fig. 9. Delay time versus gate length for 54 × 54-b multiplier.

advantage over the other multipliers in the performance/cost ratio [6], the new design of the RST multiplier makes possible a ratio in excess of 2.5.

The evidence that the 4-2 compressor can be constructed of only 48 transistors means that it has a speed advantage without a handicap in the transistor count against a series-connected pair of one-bit full adders, each consisting of 24 transistors. Furthermore, the evidence that the Booth selector can be constructed of ten transistors per bit indicates that the necessary number of transistors for generating two partial product bits in the second-order Booth encoding (20 transistors) is smaller than when directly generating four partial product bits using AND logic with the multiplicand bit and the multiplier bit at each bit position (24 transistors). This situation forces us to utilize the Booth algorithm even for a smaller multiplier such as 8 × 8-b, since this algorithm is no longer a consumer of transistors but an accelerator without hardware overhead. The measured power dissipation of this multiplier was as small as 2.23 mW/MHz at 2.5-V supply.

VI. Conclusion

In summary, we have developed sign-select Booth encoders and simplified the logic of Booth selectors that occupy a large area of the multipliers, in order to reduce the total transistor count. We also proposed a 48-transistor 4-2 compressor to save transistors without compromising in speed in the carry-save adder tree. We applied these ideas to a 54 × 54-b multiplier and fabricated a chip with 0.25-μm CMOS technology. A 54 × 54-b multiplier with 4.1-ns latency at 2.5-V supply and a 1.27 mm^2 active area was implemented. The total

transistor count was 24% less than the previously designed multiplier, and this made our device the world's smallest, fastest multiplier using CMOS technology.

The approaches described in this paper are applicable to multipliers in 3-D graphics engines with floating-point arithmetic units and those in the integer units of high-speed general-purpose microprocessors, many kinds of DSP's, and embedded controllers.

ACKNOWLEDGMENT

The authors thank K. Fujita, K. Kubota, M. Kanazawa, S. Sugatani, and S. Hijiya for their continuous encouragement.

REFERENCES

[1] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, pp. 236–240, 1951.
[2] A. Weinberger, "4-2 carry-save adder module," *IBM Tech. Discl. Bulletin*, vol. 23, Jan. 1981.
[3] V. G. Oklobdzija and D. Villeger, "Multiplier design utilizing improved column compression tree and optimized final adder in CMOS technology," in *Int. Symp. VLSI Technology, Systems and Applications*, Taipei, Taiwan, May 5–8, 1993, pp. 209–212.
[4] C. S. Wallace, "A suggestion for fast multipliers," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 14–17, Feb. 1964.
[5] ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, IEEE Computer Soc., Los Alamitos, CA, 1985.
[6] G. Goto *et al.*, "A 54 × 54-b regularly structured tree multiplier," *IEEE J. Solid-State Circuits*, vol. 27, pp. 1229–1236, Sept. 1992.
[7] Japanese Unexamined Patent Application no. 59-211138, Nov. 1984.
[8] K. Ohkubo *et al.*, "A 4.4 ns CMOS 54 × 54-b multiplier using pass-transistor multiplexer," *IEEE J. Solid-State Circuits*, pp. 251–257, vol. 30, Mar. 1995.
[9] T. Sato *et al.*, "An 8.5-ns 112-b transmission gate adder with a conflict-free bypass circuit," *IEEE J. Solid-State Circuits*, vol. 27, pp. 657–659, Apr. 1992.
[10] M. Hanawa *et al.*, "A 4.3 ns 0.3 μm CMOS 54 × 54-b multiplier using precharged pass transistor logic," in *ISSCC Dig. Tech. Papers*, Feb. 1996, pp. 364–365.
[11] J. Mori *et al.*, "A 10 ns 54 × 54-b parallel structured full array multiplier with 0.5-μm CMOS technology," *IEEE J. Solid-State Circuits*, pp. 600–606, Apr. 1991.
[12] H. Makino *et al.*, "A design of high-speed 4-2 compressor for fast multiplier," *IEICE Trans. Electron.*, pp. 538–547, Apr. 1996.
[13] H. Iino *et al.*, "A 289 MFLOPS single-chip supercomputer," in *ISSCC Dig. Tech. Papers*, Feb. 1992, pp. 112–113.
[14] M. Santoro and M. Horowitz, "SPIM: A pipelined 64 × 64 bit iterative multiplier," *IEEE J. Solid-State Circuits*, vol. 24, pp. 487–493, Apr. 1989.

Division and Square Root: Choosing the Right Implementation

PETER SODERQUIST AND MIRIAM LEESER

Peter Soderquist

Cornell University

Miriam Leeser

Northeastern University

Although frequently neglected by microprocessor designers, division and square root are critical operations. Here, we explore the four major design issues in implementing these operations: algorithm, issue rate, connectivity, and replication.

Floating-point support has become a mandatory feature of new microprocessors due to the prevalence of business, technical, and recreational applications that use these operations. Spreadsheets, CAD tools, and games, for instance, typically feature floating-point-intensive code. Over the past few years, the leading architectures have incorporated several generations of floating-point units (FPUs). However, while addition and multiplication implementations have become increasingly efficient, support for division and square root has remained uneven. The design community has reached no consensus on the type of algorithm to use for these two functions, and quality and performance of the implementations vary widely. This situation originates in skepticism about the importance of division and square root and an insufficient understanding of the design alternatives.

Division and square root have long been considered minor, bothersome members of the floating-point family. Microprocessor designers frequently perceive them as infrequent, low-priority operations, barely worth the trouble of implementing; they allocate design effort and chip resources accordingly. The survey of microprocessor FPU performance in Table 1 shows some of the uneven results of this philosophy. While multiplication requires from two to five machine cycles, division latencies range from nine to sixty. The variation is even greater for square root, which several of the designs listed do not support in hardware. This data only hints, however, at the significant variation in algorithms and topologies among the different implementations.

The error in the Intel Pentium FPU, the accompanying publicity, and the $475-million write-off illustrate some of the hazards of an incorrect division implementation.[1] But correctness is not enough; poor performance causes enough problems by itself. Though divide and square root are relatively infrequent operations in most applications, they are indispensable, particularly in many scientific programs. Compiler optimizations tend to increase the frequency of these operations by virtue of eliminating accompnaying nonarithmetic instructions.[2] This means that poor implementations disproportionately penalize code that uses them at all.[3] Furthermore, as the latency gap grows between addition/multiplication on the one hand and division/square root on the other, the latter operations increasingly become performance bottlenecks.[4]

Programmers have attempted to get around this problem by rewriting algorithms to avoid divide/square root operations, but the resulting code generally suffers from poor numerical properties, such as instability or overflow.[4-6] In short, division and square root are natural components of many algorithms, and they are best served by implementations with good performance.

Quantifying what constitutes good performance is challenging. One rule of thumb, for example, states that the latency of division should be three times that of multiplication; this figure is based on division frequencies in a selection of typical scientific applications.[7] Even if we accept this doctrine at face value, implementing division—and square root—involves much more than relative latencies. We must also consider area, throughput, complexity, and the interaction with other operations. This article explores the various trade-offs involved and illuminates the consequences of different design choices, thus enabling designers to make informed decisions.

Add-multiply configurations

Add-multiply circuits are the foundation of any FPU. Addition (which subsumes subtraction) and multiplication are not only the

Reprinted from *IEEE Micro*, Vol. 17, No. 4, pp. 56-66, July/August 1997.

most frequently occurring arithmetic operations, but together they can support all other operations required by the IEEE 754 floating-point standard.[8] We can regard all other functions, including division and square root, as additions to or enhancements of the add-multiply hardware. For these reasons, the implementation of addition and multiplication largely determines the overall performance of an FPU.

Add-multiply configurations in the latest generation of microprocessors fall into two basic categories. Figure 1 illustrates the first variety, which we have termed the independent configuration. It features add and multiply units that operate independently, reading operands from the register file and returning correctly rounded results. Although the illustration shows sharing of inputs and outputs between functional units, some implementations give each unit individual access to the register file.

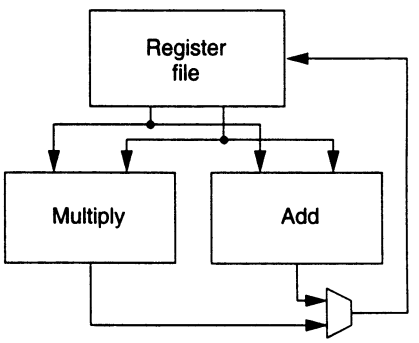
Table 1. Arithmetic performance of recent microprocessor FPUs for double-precision operands.

Design	Cycle time (ns)	Latency/throughput (cycles/cycles) $a \pm b$	$a \times b$	$a \div b$	$\sqrt{a}$
DEC 21164 Alpha AXP	2.0	4/1	4/1	22-60/22-60*	†
Hal Sparc64	6.49	4/1	4/1	8-9/7-8	†
HP PA7200	7.14	2/1	2/1	15/15	15/15
HP PA 8000	5	3/1	3/1	31/31	31/31
IBM RS/6000 Power2	13.99	2/1	2/1	16-19/15-18*	25/24*
Intel Pentium	5.0	3/1	3/2	39/39	70/70
Intel Pentium Pro	7.52	3/1	5/2	30*/30*	53*/53*
Mips R8000	13.33	4/1	4/1	20/17	23/20
Mips R10000	3.64	2/1	2/1	18/18	32/32
PowerPC 604	5.56	3/1	3/1	31/31	†
PowerPC 620	7.5	3/1	3/1	18/18	22/22
Sun SuperSparc	16.67	3/1	3/1	9/7	12/10
Sun UltraSparc	4	3/1	3/1	22/22	22/22

* Inferred from available information
† Not supported

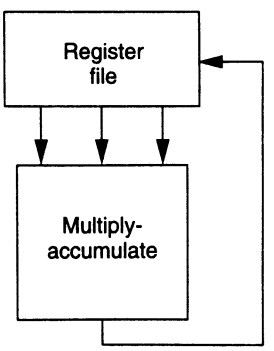
The table accompanying Figure 1a shows the typically high performance figures for this type of configuration. Frequently, the adder is actually a floating-point ALU, performing comparisons, complementation, and format conversions as well as addition and subtraction. Processors that feature independent add-multiply configurations include the DEC 21164, Hewlett-Packard PA7200, Intel Pentium Pro, and Sun UltraSparc.

Figure 1b shows the second type of add-multiply structure, known as the multiply-accumulate configuration, along with typical performance figures. This configuration combines both functions into a single atomic operation on three operands, with the multiplication of two operands followed by addition of the product and the third operand. Typically, this unit maintains the product at a very high precision and adds without intermediate rounding. The motivation for the multiply-accumulate configuration is that a large proportion of scientific computations, such as series and matrix operations, involve many instances of multiplication followed immediately by addition. The IBM RS/6000 series, the Mips R8000, the HP PA8000, and the Hal Sparc64 use add-multiply structures of this type.



Operation	Latency	Throughput
Addition	2	1
Multiplication	2	1

(a)



Operation	Latency	Throughput
Multiply-accumulate	2	1

(b)

Figure 1. Typical independent add-multiply configuration (a) and typical multiply-accumulate configuration (b).

Design issues

Given the add-multiply infrastructure, there are many ways to incorporate division and square root functionality into an FPU. The following is a brief introduction to the major design decisions, which we will explore in greater detail in subsequent sections. The primary issues fall into four categories:

- *Algorithm.* The choice of a divide/square root algorithm is by far the most important design decision. It involves a series of smaller decisions, and once made, largely determines the subsequent range of choices and the area

and performance of the implementation in general.

- *Issue rate.* Although some chips execute floating-point instructions strictly in series, many of the most recent microprocessors can issue up to two floating-point instructions per cycle. Therefore, the next issue worthy of attention is the impact of floating-point issue rate on the efficiency of divide/square root implementations.
- *Connectivity.* Closely related to issue rate is the manner in which the divide/square root hardware interfaces with the rest of the FPU, either with independent access to the register file, or coupled to or integrated with one of the other functional units.
- *Replication.* Finally, it is becoming more common for microprocessors to duplicate floating-point functional units, including the divide/square root hardware. The performance and area effects of such replication deserve closer consideration.

These issues are not independent but interact with each other in complex and nonobvious ways. The following sections explain the connections between the issues, while trying to isolate and quantify their effects on the design trade-offs.

Simulation

To provide a uniform, quantitative foundation for the analysis of the design trade-offs, we implemented an FPU-level simulator that executes a benchmark based on Givens rotations. (For a description of Givens rotations, please see the supplement to this article on the World Wide Web at http://www.computer.org/pubs/micro/micro.htm.) Given an FPU configuration, including its structure and performance on individual operations, the simulator calculates the number of cycles required to transform an arbitrary matrix into triangular form using Givens rotations. The simulations use a standard set of test matrices ranging in size from 10 × 10 elements to 200 × 100 elements.[4]

We chose the Givens rotation algorithm because it is rich in divide and square root operations, and because it is a real program with important numerical applications. The primary task of many scientific programs is the solution of partial differential equations. Givens rotations are a central component of several frequently used solution techniques. They also feature prominently in signal processing algorithms, and the rotation and projection algorithms common in graphics and solid modeling use similar combinations of operations.

The Givens rotation algorithm is not typical in its use of divide/square root operations. In fact, there are probably few other real applications whose performance is so dependent on the efficiency of division and square root. Nevertheless, what it lacks in representativeness, it makes up for in importance. Finally, as a kind of divide/square root torture test, Givens rotations clearly illustrate the effects of different implementation choices.

Divide/square root algorithms

There are many possible algorithms for computing division and square root, but only a subset are currently practical for microprocessor implementation. These fall into the two categories of multiplicative and subtractive algorithms.[9] Multiplicative methods use hardware integrated with the floating-point multiplier and have low to moderate latencies, while subtractive methods generally employ separate circuitry and have low to high latencies. We consider only the case that implements division and square root in hardware using the same algorithm. We cover the arguments for this decision in more detail elsewhere;[4] in summary, division and square root tend to have very similar implementations and allow for extensive hardware sharing. Furthermore, this is the most common case in actual practice.

Multiplicative techniques. Multiplicative algorithms compute an estimate of the quotient or square root using iterative refinement of an initial guess. They reduce divide and square root operations into a series of multiplications, subtractions, and bit shifts. In addition, multiplicative algorithms typically converge at a quadratic rate, which means the number of accurate digits in the estimate doubles at each iteration. The two techniques used in recent microprocessors are the Newton-Raphson method and Goldschmidt's algorithm. With many features in common, they differ primarily in operation ordering, which affects their mapping onto pipelined hardware.

Newton-Raphson method. An algorithm with a long history, the Newton-Raphson method[10,11] has been implemented in many machines, including the IBM RS/6000 series. To find a/b, set the "seed value" x_0 to an estimate of $1/b$ and iterate over

$$x_{i+1} = x_i \times (2 - b \times x_i)$$

until x_{i+1} is within the desired precision of $1/b$. Then $a \times x_{i+1} \approx a/b$.

For square root computation, set the seed to approximate $1/\sqrt{a}$ and refine it with the iteration

$$x_{i+1} = 1/2 \times x_i \times (3 - a \times x_i^2).$$

The product $a \times x_{i+1}$ yields an approximation of $\sqrt{a}$.

Goldschmidt's algorithm. Recent implementations of this algorithm[11] have included the Sun SuperSparc and arithmetic coprocessors from Texas Instruments. To compute $a/b = x_0/y_0$, iteratively calculate

$$x_{i+1} = x_i \times r_i \text{ and } y_{i+1} = y_i \times r_i.$$

Here, $r_i = 2 - y_i$; then $y_i \rightarrow 1$ and $x_i \rightarrow a/b$. Both x_0 and y_0 should be prescaled by a seed value approximating $1/b$ to ensure rapid convergence. To compute $\sqrt{a}$, set $x_0 = y_0 = a$, prescale by an estimate of $1/\sqrt{a}$, and iterate over

$$x_{i+1} = x_i \times r_i^2 \text{ and } y_{i+1} = y_i \times r_i,$$

where $r_i = (3 - x_i)/2$, such that

$$x_{i+1}/y_{i+1}^2 = x_i/y_i^2 = 1/a.$$

Then $x_i \rightarrow 1$, and hence $y_i \rightarrow \sqrt{a}$.

Implementations. Multiplicative divide/square root implementations generally take the form of modifications to the floating-point multiplier. This is because divide and square

root functionality alone do not justify the area required for a second multiplier. The designers of the IBM RS/6000 FPU chose the Newton-Raphson method as being best suited to the multiply-accumulate structure. In independent add-multiply configurations, Goldschmidt's algorithm is a natural choice for pipelined multipliers. Since the numerator and denominator operations are independent, clever scheduling allows the multiplier to run at maximum efficiency. The Newton-Raphson method suffers from dependencies between successive operations, which hobble pipelining.

The block diagram in Figure 2 shows one possible implementation, namely an independent floating-point multiplier modified for Goldschmidt's algorithm.

Although the details will vary from case to case, most multiplicative implementations feature one or more of the following hardware enhancements:

- *Extra routing and storage.* These are required to render divide and square root as atomic operations independent of the register file.
- *Seed value lookup tables.* An algorithm's execution time relates directly to the accuracy of the initial guess. An accurate double-precision value can be produced in three iterations with an 8-bit seed, while a 16-bit seed can bring the number of iterations down to two. Unfortunately, the lookup table for the latter case must store 512 times as many bits.[4]
- *Constant subtraction/shifting logic.* For independent multipliers, supporting the algorithm's nonmultiply operations directly improves performance and maintains independence from the floating-point adder. Multiply-accumulate structures have these capabilities built in.
- *Last-digit rounding support.* Values derived from reciprocal estimates, as in the Newton-Raphson method and Goldschmidt's algorithm, have an inherent error that can lead to inaccuracies in the last result digit. The several strategies for solving this nontrivial problem carry associated area and performance trade-offs.[4] Multiply-accumulate units can avoid extra hardware at the expense of extra operations.

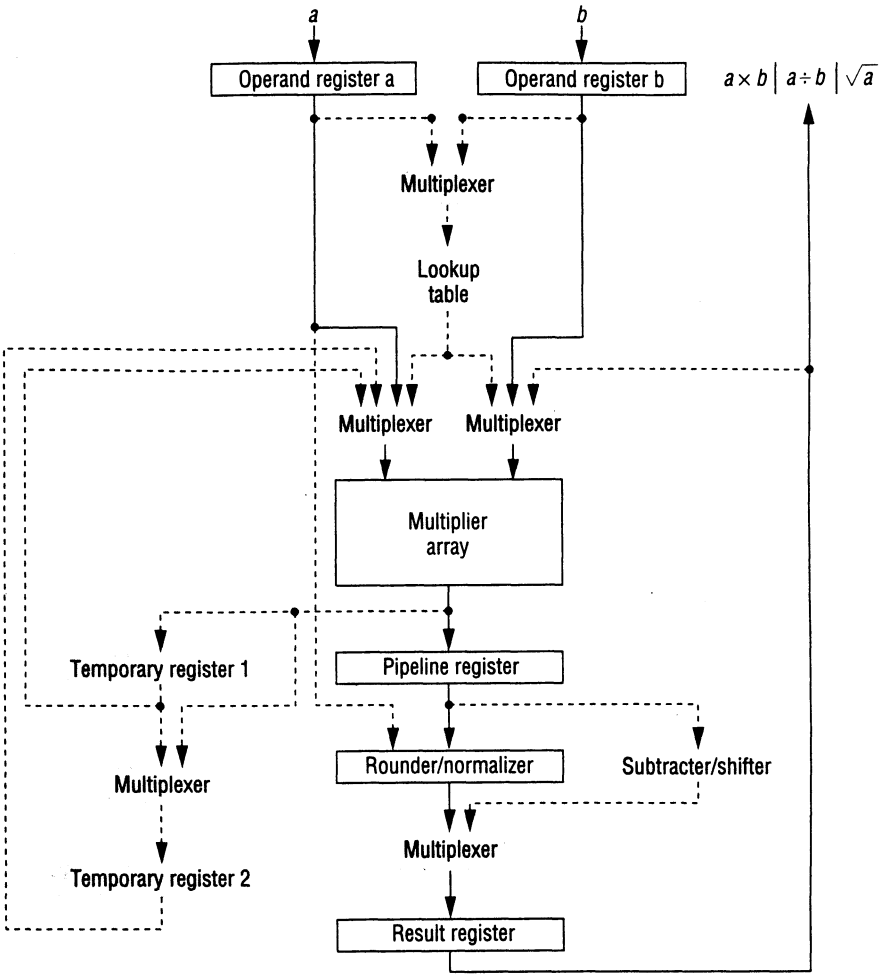


Figure 2. A floating-point multiplier enhanced for multiplicative divide/square root computation. Shading indicates new components, and dashed lines indicate new routing required for divide-square root.

Multiplicative divide/square root implementations can incur a penalty beyond the area needed for extra logic, storage, and routing. Since some of the required modifications lie on the multiplier's critical path, they can negatively affect the latency of multiplication itself.

Subtractive techniques. The subtractive divide/square root algorithms employed in current microprocessors can generally be classified as SRT methods. These are named for D. Sweeny, J.E. Robertson, and K.D. Tocher, who independently developed very similar algorithms for division.[10] Subtractive division and square root compute the quotient q or square root s directly, one digit at a time. The conventional procedure for long division by hand is an algorithm of this type. In practice, subtractive algorithms typically use redundant representations of values and treat groups of consecutive bits as higher-radix digits to enhance performance.[12] For example, a radix-4 divider scans successive pairs of bits as single digits—interpreting two radix-2 values as a single radix-4 one.

Algorithm. For division, let $q[j]$ be the partial quotient at step j (where $q[n] = q$), and let $w[j]$ be the residual or partial remainder. The goal of the algorithm is to find the sequence of quotient digits that minimizes the residual at each step. To compute $q = x \div d$ for n-digit, radix-r values, set $w[0] = x$ and evaluate the recurrence

$$w[j+1] = rw[j] - dq_{j+1},$$

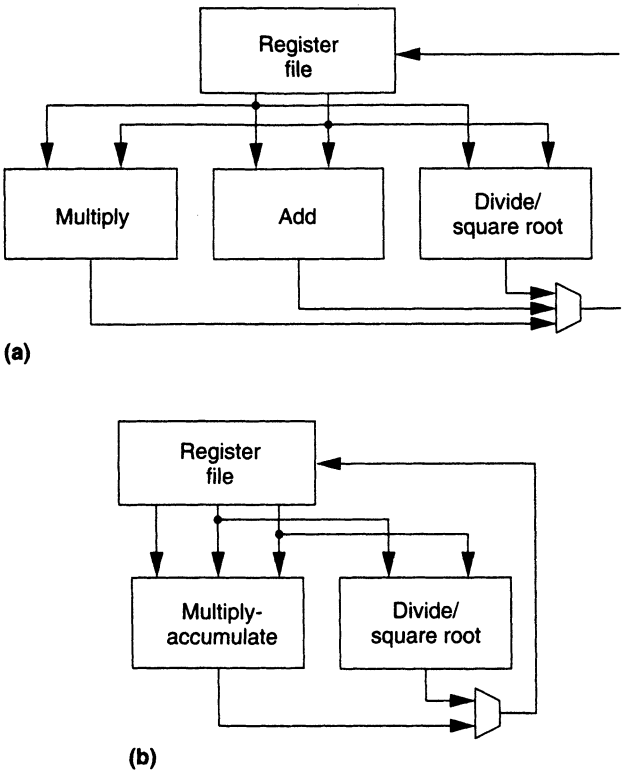


Figure 3. FPU topologies with subtractive divide/square root functional units and independent add-multiply (a) or multiply-accumulate configurations (b).

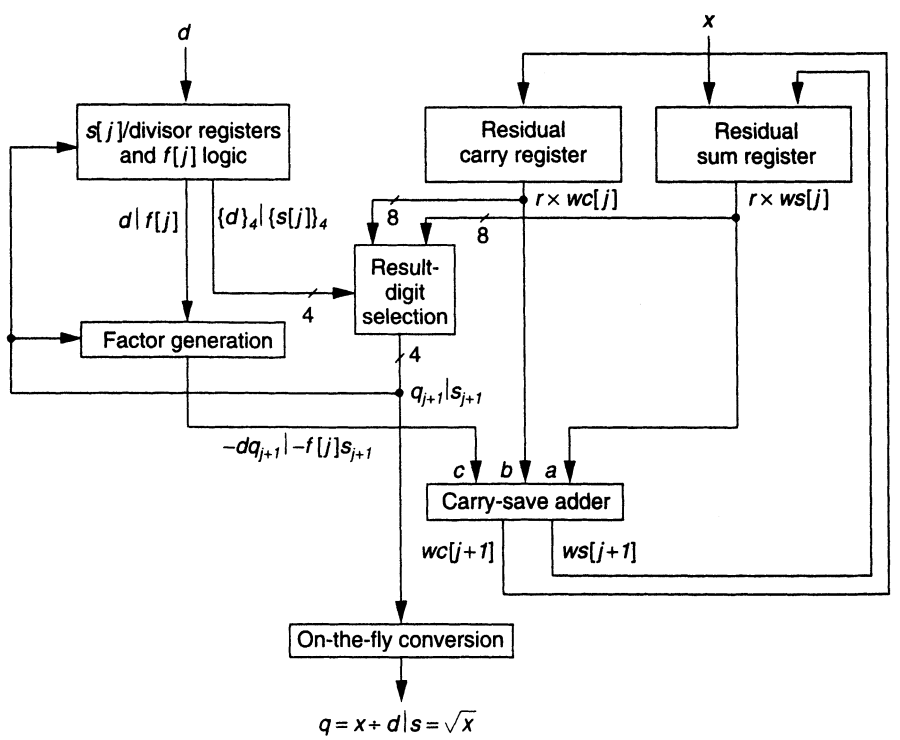


Figure 4. Radix-4 SRT divide/square root unit.

where q_{j+1} is the $(j + 1)$th quotient digit.

For square root computation, let $s[j]$ denote the jth partial root. To find $s = s[n] = \sqrt{x}$, set $w[0] = x$ and evaluate

$$w[j + 1] = rw[j] - 2s[j]s_{j+1} - s_{j+1}^2 r^{-(j+1)}.$$

Define $f[j] = 2s[j] - s_{j+1}r^{-(j+1)}$; then

$$w[j + 1] = rw[j] - f[j]s_{j+1}$$

has the same form as the division recurrence. In practice, $f[j]$ is simple to generate, which facilitates combined division and square root implementations.

Implementations. Subtractive divide/square root implementations generally rely on dedicated hardware that executes in parallel with the remainder of the FPU, as shown in Figure 3. A minority of designs, such as the IBM/Motorola PowerPC line and the Mips R4000 series, use enhancements of the addition hardware. However, a separate functional unit offers superior performance, since the multiplication and addition hardware can continue to operate on independent instructions while the other unit computes the quotient or root. If multiplication hardware, for example, supports divide and square root, then these operations—which typically have long latencies—tie up the unit, holding up subsequent multiply instructions until the quotient or root is available.

Figure 4 shows a basic radix-4 divide/square root unit, including most of the features critical to its high performance. The residual is stored in redundant form as vectors of sum and carry bits; this enables the use of a low-latency carry-save adder to calculate the subtraction in the recurrence. The result-digit selection table returns the next quotient/square root digit on the basis of the residual and divisor/partial root values; the redundant result-digit set allows truncated input values, keeping the table small and fast. Factor generation logic keeps all possible multiples of d or $f[j]$ and every result digit (qj or $sj \in \{-2, -1, 0, 1, 2\}$) available at all times for immediate multiplexer selection. Finally, the on-the-fly conversion logic[12] operates concurrently with quotient computation and off of the critical path. This maintains updated values of $s[j]$ and $f[j]$ and incrementally converts the partial result from redundant into conventional representation.

While radix-2 divide/square root units operate on one bit at a time, radix-4 implementations retire two bits of the result at every iteration. Higher-radix

units process even larger groups of bits at once. Unfortunately, for radices greater than 8, the latency and area requirements of result digit selection and factor generation become prohibitive. To circumvent these effects, we can combine lower-radix stages into a single higher-radix unit. We can obtain radix-16 division, for example, by overlapping two radix-4 dividers, with only a modest increase in area and iteration delay over radix-4 alone.[12]

Certain methods achieve even higher radix operations; the majority of these are restricted to theoretical treatment or experimental implementations. Ercegovac, Lang, and Montuschi summarize and compare several of these approaches.[13]

One way to improve the performance of any subtractive implementation is to boost the number of iterations per machine cycle, if possible. In the HP PA7200, for example, the divide/square root unit's low iteration delay means that it can cycle at twice the system clock rate. Another technique that offers extremely low latencies is self-timing, whereby a functional unit operates asynchronously with respect to the rest of the FPU, completing each iteration at the highest rate its internal logic permits.[14] The Hal Sparc64 implements a version of self-timed division.

Table 2. Divide/square root performance of independent implementations.

Algorithm	Latency (cycles)		Execution time improvement (%)		
	Divide	Square root	Maximum	Minimum	Average
8-bit seed Goldschmidt	9	13	0.0	0.0	0.0
16-bit seed Goldschmidt	7	10	9.9	1.6	5.0
Radix-4 SRT	15	15	20.9	7.2	15.2
Radix-16 SRT	8	8	46.0	7.2	23.4

Table 3. Divide/square root performance of multiply-accumulate implementations.

Algorithm	Latency (cycles)		Execution time improvement (%)		
	Divide	Square root	Maximum	Minimum	Average
8-bit seed Newton-Raphson	19	22	0.0	0.0	0.0
16-bit seed Newton-Raphson	14	17	19.7	4.9	11.6
Radix-4 SRT	15	15	69.4	22.4	48.3
Radix-16 SRT	8	8	125.7	22.5	68.8

Performance impact. To determine how the algorithms compare in terms of performance, we matched each add-multiply configuration with a selected set of divide/square root implementations and executed the Givens rotation benchmark on the standard matrix data set. For each configuration (independent and multiply-accumulate) we assumed a floating-point issue rate of one instruction per cycle. We closely modeled the FPU structures and performance figures for individual operations on real examples. We compared the following divide/square root implementations:

- *8-bit seed multiplicative.* This is a baseline multiplicative implementation with an 8-bit seed lookup table, as used in many actual chips. The independent configuration uses Goldschmidt's algorithm, while the multiply-accumulate structure implements the Newton-Raphson method.
- *16-bit seed multiplicative.* As before, we matched this algorithm to the add-multiply configuration; the larger lookup table reduces the latency of divide/square root computation.
- *Radix-4 SRT.* The subtractive equivalent of the 8-bit seed case, this is a basic, practical implementation using a separate functional unit.
- *Radix-16 SRT.* Overlapping radix-4 units provide lower-latency operations.

The first set of experiments matches the independent configuration of Figure 1, which we modeled on the HP PA7200, with the divide/square root implementations just listed. Table 2 gives division and square root performance for each case. The radix-4 case uses the actual latency figures from the HP PA7200. This implementation computes four quotient digits per machine cycle by clocking the divide/square root unit at double speed. For radix-16 operations, the figures assume that the configuration can accommodate the slightly longer cycle time of the overlapped implementation without difficulty. We based the Goldschmidt's algorithm latencies on the Texas Instruments implementations parameterized by the latency of multiplication and the accuracy of the seed value.

Table 2 also summarizes the relative performance of the different implementations on the Givens rotation benchmark, normalized to the 8-bit seed Goldschmidt case. The subtractive implementations clearly dominate the multiplicative ones, even in cases where the latter have superior latencies. It is the ability to execute in parallel with the rest of the FPU that gives the radix-4 and radix-16 units their decisive performance advantage.

The second set of experiments pairs the multiply-accumulate structure shown in Figure 2 with the standard set of divide/square root implementations. This configuration and the first divide/square root implementation in Table 3 are based on the IBM RS/6000 series FPU, which uses specially adapted versions of the Newton-Raphson method. The 16-bit seed version uses these same algorithms but assumes a reduced number of iterations. As for the subtractive implementations, we simply carried them over from the HP PA7200 case. Not only does this permit a more uniform comparison, but it seems like a feasible implementation, since the RS/6000 actually has a longer cycle time than the PA7200 while using a comparable fabrication technology.

The Givens rotation results, also shown in Table 3, display the same pattern as for the independent add-multiply case, albeit with even greater contrast. The Newton-Raphson

Table 4. Area comparison of two divide/square root implementations.

Device	Algorithm	Chip area (mm^2)	Transistor count	Divider/square root area (mm^2)
Weitek 3364	Radix-4 SRT	95.2	165,000	4.8
TI 8847	8-bit seed Goldschmidt	100.8	180,000	6.3

Table 5. Relative cost of different divide/square root implementations.

Algorithm	Area factor
8-bit seed multiplicative	1.0
16-bit seed multiplicative	22
Radix-4 SRT	1.5
Radix-16 SRT	2.2

operations' longer latencies mean that the subtractive implementations perform even better by comparison. Note also the significant improvement that the radix-16 implementation affords over the radix-4 case.

Area considerations. There are vast differences in the implementation styles of multiplicative and subtractive methods. The former are primarily enhancements of existing circuitry, while the latter inhabit completely new hardware. Furthermore, the core operations are entirely different. In the interest of an objective evaluation, it is essential to compare the circuit area required by these very distinct approaches.

Estimating area in a way that yields a fair comparison between chips is difficult because of basic differences in the implementation technologies. Nevertheless, it is possible to give some basis for evaluating the different implementations. Table 4 compares the size of the hardware dedicated to division in the Weitek 3364 and Texas Instruments 8847 arithmetic coprocessors; we based the figures on measurements of microphotographs.[11]

The chips have similar die sizes and device densities, and we can assume the feature sizes are comparable as well. Although the multiplication algorithms are different, both have two-pass arrays that take up approximately 22% of the chip area. In short, apart from their divide/square root implementations, these two chips have a lot in common. However, the area devoted to division hardware is little more than 6% of the chip size in either case. Also, the relative area requirements differ by 1.5%. These figures represent only two particular designs whose implementation technology is by now somewhat out of date. However, they suggest that 8-bit seed Goldschmidt and radix-4 SRT implementations are both economical, and that the area differences between them can be kept small.

An alternative approach uses standard-cell technology to estimate the areas of different implementations, including the more sophisticated options.[4] Table 5 shows the results of this study, with the values normalized to the size of the 8-bit seed Goldschmidt variant. According to these results, a radix-16 SRT unit need only be 45% larger than a radix-4 design in the same technology. A 16-bit seed Goldschmidt implementation, by contrast, could be over 20 times larger than the 8-bit seed implementation, and 10 times larger than the radix-16 SRT unit. This calls into question the practicality of such a solution, especially since it leads to a relatively small improvement in latency as evidenced by Tables 2 and 3.

Summary. Multiplicative implementations can provide low latency and lower cost than subtractive ones. However, their overall performance on our benchmark is inferior, mainly because the multiplication hardware is responsible for additional nonpipelined, multicycle operations. With the multiply-accumulate configuration, the same pipeline must accommodate addition, multiplication, division, and square root. Although 8-bit implementations tend to require less area than radix-4 hardware, 16-bit lookup tables dwarf even radix-16 implementations. Furthermore, the incredibly expensive upgrade from an 8-bit seed table to a 16-bit one offers only a modest reduction in divide/square root latency, and a downright meager improvement in benchmark execution time.

Subtractive divide/root implementations provide superior benchmark performance at a reasonable cost, for both independent and multiply-accumulate configurations. This is primarily due to the fact that subtractive hardware operates independently and does not tie up other FPU resources. The parallel operation also means that improvements in divide/square root latency have a more decisive impact than in multiplicative implementations. Upgrading from radix-4 to radix-16 can improve performance by as much as 21% for the independent case or 56% for the multiply-accumulate structure, at a cost of 33% more area.

Subtractive implementations also fit in nicely with the current trend toward decoupled superscalar processors with high instruction issue rates and growing transistor budgets. We conclude that implementations of subtractive divide/square root algorithms are the most sensible choice, and focus on them for the remainder of the discussion.

Issue rate

The simulations in the previous section assumed an issue rate of one floating-point instruction per machine cycle. This situation holds in such processors as the PowerPC series, where the FPU is a single pipeline, and the Intel Pentium Pro, where the arithmetic functional units share a single set of inputs and outputs. However, many recent machines, including the Mips 10000 and Sun UltraSparc, can generate up to two floating-point instructions at once.

Dual-issue operation only makes sense when there are at least two independent functional units. Furthermore, if one of the two functional units is a dedicated divide/square root unit, dual issue is not worthwhile because these instructions are not frequent enough to keep a separate unit busy. By this reasoning, dual floating-point instruction issue is appropriate for an FPU with an independent adder, multiplier, and divide/square root unit, but not one with a multiply-accu-

mulate structure and separate divide/square root unit.

Performance impact. To explore the interaction of higher issue rates and divide/square root performance, we ran a series of experiments using the Givens rotation benchmark. For the reasons given earlier, we restricted our focus to independent add-multiply configurations with independent, subtractive divide/square root implementations. The variables are the instruction issue rate (single or dual) and the algorithm of the divide/square root implementation (radix-4 or radix-16). Performance figures for individual operations are the same as from previous experiments.

From the results in Table 6, it is apparent that increasing the floating-point instruction issue rate leads to a genuine performance boost. This effect tends to overshadow the effects of divide/square root latency. The dual-issue radix-4 case outperforms the single-issue radix-16 one, even though the latter is faster on individual operations. Also, the difference between the dual-issue radix-4 and radix-16 cases is much less significant than the contrast between cases with the same algorithm but differing issue rates. This is another instance, not unlike the contest between multiplicative and subtractive algorithms, where parallelism wins over latency. By keeping all of the units busier, especially the adder and multiplier, dual-instruction issue softens the impact of division latency.

We also performed a set of simulations to determine how dual-instruction issue affects the performance balance between multiplicative and subtractive methods. As shown in Table 7, multiple instruction issue strengthens the performance advantage of subtractive implementations. This is what one would expect, given the increased parallelism of operations afforded by a separate functional unit for divide/square root operations. These data support the claim that subtractive methods are better suited to superscalar implementations.

Area considerations. The choice of how many floating-point instructions to dispatch per cycle is a much larger issue than divide/square root implementation alone, and probably not entirely up to the floating-point designer. Nevertheless, if the option is available, there are several issues to consider. Obviously, there must be at least as many functional units as the maximum number of instructions issued per cycle. Also, allocating one instruction per cycle to divide/square root operations alone is a waste of resources. In addition, there is a cost to boosting the floating-point issue rate that we cannot readily quantify. Routing must be added to deliver operands and return results. The register file needs extra ports, which add to its area and access time. Finally, there are the necessary changes to the dispatch logic, which affect the processor as a whole.

Table 6. Effect of floating-point instruction issue rate on benchmark performance; execution time improvement for an independent configuration.

Algorithm	Issue rate	Execution time improvement (%) Maximum	Minimum	Average
Radix-4 SRT	Single	0.0	0.0	0.0
Radix-16 SRT	Single	26.9	0.0	7.0
Radix-4 SRT	Dual	50.0	5.5	35.4
Radix-16 SRT	Dual	52.1	22.3	44.4

Table 7. Effect of algorithm choice on benchmark performance for dual-issue configurations.

Algorithm	Execution time improvement (%) Maximum	Minimum	Average
8-bit seed Goldschmidt	0.0	0.0	0.0
16-bit seed Goldschmidt	11.4	2.3	6.2
Radix-4 SRT	37.2	4.5	20.1
Radix-16 SRT	46.7	14.1	28.9

Connectivity

In an FPU with a dual-issue, independent add-multiply configuration, there are several ways to connect subtractive divide/square root hardware. We can either package divide/square root circuitry as an independent functional unit as in the Sun UltraSparc, share ports with the adder as in the DEC 21164, or share ports with the multiplier as in the Mips R10000. Figure 5 (next page) illustrates the possible configurations. Each of these implementations computes division and square root concurrently with other FPU operations. However, certain configurations lead to contention for shared input and/or output ports.

Multiplicative divide/square root hardware is always integrated with the multiplier, so the connectivity issue is moot for those types of implementations. Sharing subtractive divide/square root unit ports with a multiply-accumulate unit is equivalent to single-issue FPU, a case we have already covered.

Performance impact. At first glance, sharing functional unit ports between divide/square root and other operations seems sensible, since addition and multiplication occur much more frequently. It is important, however, to minimize collisions between divide/square root and the host operations. The simulations we performed explore the impact of different interconnection schemes on benchmark performance.

In every case, we assumed an independent add-multiply configuration and dual floating-point instruction issue. The variables are the algorithm (radix-4 or radix-16) and the connectivity (independent, adder-coupled, or multiplier-coupled). In cases where operations share functional units, we pushed divide and square root operations back one cycle in the event of a port conflict.

The simulation results, given in Table 8, are somewhat

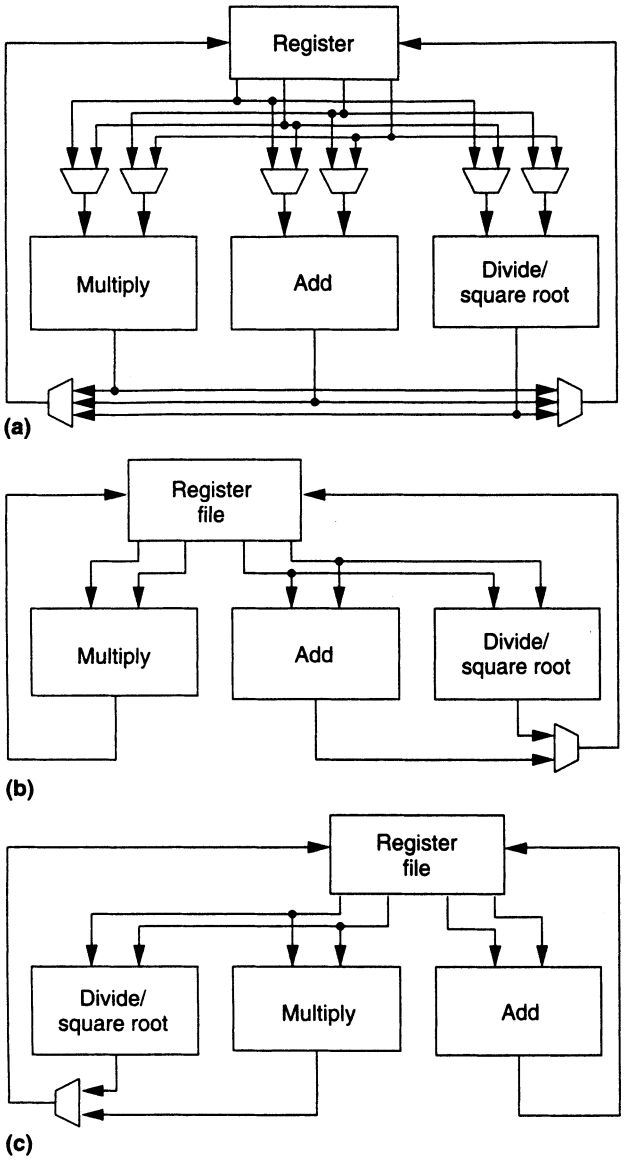


Figure 5. Dual-issue, independent add-multiply configurations with independent (a), adder-coupled (b), and multiplier-coupled (c) divide/square root units.

counterintuitive. First, for both algorithms, the independent and adder-coupled cases show exactly the same performance. This is a feature of the Givens rotations algorithm, which has approximately twice as many multiplies as adds, and, therefore, allocates many free issue slots to the adder. The adder-coupled case can take advantage of these openings, yielding the same performance as the independent case. The multiply-coupled configuration is hampered by frequent collisions between multiply and divide/square root operations, but the effects are relatively mild, especially for the radix-4 cases. With the shorter latency of radix-16 operations, the collisions have comparatively greater impact on performance, but still less than 7% on average.

Although the Givens rotation benchmark favors the adder-coupled design, we must not take its near 2:1 ratio of multiply to add operations as typical. In a survey of floating-point operations in the SPEC92fp benchmark suite, the multiply:add ratio is closer to 2:3.[2] This supports the intuitive suggestion that addition rather than multiplication dominates many applications, and that multiplier-coupled divide/square root is therefore the most appropriate choice. In every other design decision we have covered so far, improving the performance of the benchmark is not in conflict with enhancing the speed of all programs; this case is an exception.

Area considerations. The primary cost of a separate, independent divide/square root unit is the extra routing required, including the extra buses and the multiplexing/ selection logic. Compare the FPU routing complexity of the independently connected divide/square root unit in Figure 5a with the adder- and multiplier-coupled cases in Figures 5b and 5c. When the divide/square root unit is coupled to one of the other functional units, we can readily split the two sets of ports on the register file between the two clusters of components. Connecting two sets of ports to three independent units is considerably more complicated. Similarly, independent divide/square root units complicate the instruction dispatch logic by increasing the number of possible paths and destinations for instructions.

Replication

Several microprocessor manufacturers have recently attempted to boost numerical performance by replicating floating-point functional units. The IBM RS/6000 Power2 and Mips R8000 feature dual multiply-accumulate units, and the Mips R10000 couples its independent multiplier with two other units for divide and square root. The HP PA8000 FPU consists of two multiply-accumulate units and two divide/square root units. Increasingly compact technology and higher maximum issue rates facilitate this trend.

There are also many different ways to replicate floating-point functionality and connect it with the rest of the system. For processors with FPUs organized as single, atomic pipelines or blocks, such as the PowerPC and IBM RS/6000 series, the most natural alternative is to replicate the entire

Table 8. Effect of divide/square root connectivity on benchmark performance; execution time improvement for an independent configuration.

Algorithm	Connectivity	Execution time improvement (%) Maximum	Minimum	Average
Radix-4 SRT	Multiplier-coupled	0.0	0.0	0.0
Radix-4 SRT	Adder-coupled	4.4	1.4	2.8
Radix-4 SRT	Independent	4.4	1.4	2.8
Radix-16 SRT	Multiplier-coupled	12.1	0.1	3.8
Radix-16 SRT	Adder-coupled	26.3	2.3	10.6
Radix-16 SRT	Independent	26.3	2.3	10.6

Table 9. Summary of divide/square root design decisions for different architectures.

Design	Add-multiply configuration	Division/square root design decisions: Algorithm	FP issue rate (instr./cycle)	Connectivity	Units (no.)
DEC 21164 Alpha AXP	Independent	SRT	2	Adder-coupled	1
Hal Sparc64	MAC*	SRT	2	Independent	1
HP PA7200	Independent	SRT	2	Independent	1
HP PA8000	MAC	SRT	2	MAC-coupled	2
IBM RS/6000 Power2	MAC	Newton-Raphson	2	Integrated	2
Intel Pentium	Independent	SRT	1	Adder-coupled	1
Intel Pentium Pro	Independent	SRT	1	Independent	1
Mips R8000	MAC	Multiplicative	2	Integrated	2
Mips R10000	Independent	SRT	2	Multiplier-coupled	2
PowerPC 604	MAC	SRT	1	Integrated	1
PowerPC 620	MAC	SRT	1	Integrated	1
Sun SuperSparc	Independent	Goldschmidt	1	Multiplier-integrated	1
Sun UltraSparc	Independent	SRT	2	Independent	1

* Multiply-accumulate

unit. Decoupled superscalar designs like the Intel Pentium Pro and Sun SuperSparc are more at liberty to add individual units, such as an extra divide/square root unit.

Performance impact. There are many possible ways to analyze how replicating floating-point hardware affects divide/square root performance. We considered one example particularly worthy of examination—namely, the duplication of divide/square root hardware in a fully independent FPU. However, from a preliminary investigation, we concluded that the addition would have no effect on Givens rotation benchmark performance, unless we heavily modified the method of scheduling operations. Even with a much more aggressive schedule, we expect the effects would be insignificant.

Addressing the performance impact of replicating floating-point functionality in general is beyond the scope of this investigation. However, since the Givens rotation benchmark represents an unusually high level of divide/square root use, replicating the hardware for these functions alone seems to offer little potential benefit.

Area considerations. It is a more efficient use of area to improve a slow divide/square root unit than to replicate it. For example, in a dual-issue, independent FPU, upgrading from radix-4 to radix-16 gives a maximum improvement of 22.5% and an average improvement of 7.5% on the Givens rotation benchmark. Duplication of the radix-4 unit, on the other hand, gives no appreciable performance benefit. The upgrade to radix-16 costs only 45% of the radix-4 area, as opposed to 100% for replication, not including all of the extra routing and any changes to the register file or instruction issue logic.

FLOATING-POINT DIVISION AND SQUARE ROOT

are important operations warranting fast, efficient implementations. Table 9 summarizes the design decisions made in recent microprocessors.

Subtractive divide/square root implementations offer latencies competitive with the fastest multiplicative ones, but with a considerably lower consumption of chip area. More significantly, by operating concurrently with the remainder of the FPU, subtractive hardware provides a potentially significant boost in performance, as our simulations on the Givens rotation benchmark evidence. This performance advantage holds true for both multiply-accumulate and independent add-multiply configurations. Unlike multiplicative divide/square root hardware, which serializes computation, subtractive implementations are well suited to exploit decoupled superscalar microarchitectures and issue rates of multiple floating-point instructions per cycle.

For subtractive implementations, increasing the instruction issue rate from one to two instructions per cycle can also dramatically improve performance—by over 35% on average for the Givens rotation benchmark. As long as the implementation does not squander the extra instruction on divide/square root operations alone, this seems like a worthwhile improvement, taking into account the cost of routing and upgrading the register file and instruction issue logic.

The cost and complexity of adding a fully independent divide/square root unit to a dual-issue, independent add-multiply configuration are considerable. Therefore, coupling this functionality to one of the existing structures by sharing connections to the register file seems like a reasonable economizing measure. The worst-case performance impact seems modest compared with the possible explosion in routing costs and the additional scheduling complications incurred. The balance of the evidence suggests that the multiplier is the best candidate for sharing ports with the divide/square root unit.

The merits of replicating floating-point functional units,

however, remain to be seen. Certainly, even with highly parallel algorithms, one can expect significantly less than twice the performance from a doubling of hardware. In particular, duplicating divide/square root hardware alone appears to hold few advantages, even for the Givens rotation benchmark with its heavy use of these operations.

The growing popularity of 3D graphics and multimedia applications, and the allocation of the computational burden to the CPU (as for example, in Intel's MMX technology), mean ever-increasing demands for microprocessor arithmetic performance. The optimal integration of multimedia functionality with existing floating-point infrastructure is an area ripe for future research.

Acknowledgments

The National Science Foundation partly supported this research under contract CCR-9257280. Peter Soderquist's support included a fellowship from the National Science Foundation. Miriam Leeser's support included an NSF Young Investigator award.

References

1. "The Pentium Papers," Mathworks, Inc., Natick, Mass., Nov. 1994, http://www.mathworks.com/README.html.
2. S.F. Oberman and M.J. Flynn, "Design Issues in Floating-Point Division," Tech. Report CSL-TR-94-647, Stanford University, Departments of Electrical Engineering and Computer Science, Stanford, Calif., Dec. 1994.
3. S.E. McQuillan, J.V. McCanny, and R. Hamill, "New Algorithms and VLSI Architectures for SRT Division and Square Root," *Proc. 11th IEEE Symp. Computer Arithmetic,* IEEE, Piscataway, N.J., 1993, pp. 80-86.
4. P. Soderquist and M. Leeser, "Area and Performance Tradeoffs in Floating-Point Division and Square Root Implementations," *ACM Computing Surveys,* Vol. 28, No. 3, Sept. 1996, pp. 518-564.
5. S.E. McQuillan and J.V. McCanny, "A VLSI Architecture for Multiplication, Division, and Square Root," *Proc. 1991 Int'l Conf. Acoustics, Speech and Signal Processing,* IEEE, 1991, pp. 1205-1208.
6. W. Kahan, *Using MathCAD 3.1 on a Mac,* unpublished article, Aug. 1994; available upon request from author.
7. B.K. Bose et al., "Fast Multiply and Divide for a VLSI Floating-Point Unit," *Proc. Eighth IEEE Symp. Computer Arithmetic,* IEEE, 1987, pp. 87-94.
8. *ANSI/IEEE Std. 754–1985, Binary Floating-Point Arithmetic,* IEEE, New York, 1985.
9. P. Soderquist and M. Leeser, "An Area/Performance Comparison of Subtractive and Multiplicative Divide/Square Root Implementations," *Proc. 12th IEEE Symp. Computer Arithmetic,* IEEE, 1995, pp. 132-139.
10. N.R. Scott, *Computer Number Systems and Arithmetic,* Prentice Hall, Englewood Cliffs, N.J., 1985.
11. D. Goldberg "Appendix A, Computer Arithmetic," in J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, San Francisco, 1990.
12. M.D. Ercegovac and T. Lang, *Division and Square Root: Digit Recurrence Algorithms and Implementations,* Kluwer Academic Publishers, Norwell, Mass., 1994.
13. M.D. Ercegovac, T. Lang, and P. Montuschi, "Very High Radix Division with Selection by Rounding and Prescaling," *Proc. 11th IEEE Symp. Computer Arithmetic,* IEEE, 1993, pp. 112-119.
14. T.E. Williams, "A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider," *IEEE J. Solid-State Circuits,* Vol. 26, No. 11, Nov. 1991, pp. 1651-1661.

167 MHz Radix-8 Divide and Square Root Using Overlapped Radix-2 Stages

J. Arjun Prabhu and Gregory B. Zyner

SPARC Technology Business, Sun Microsystems, Inc.
Mountain View, California

Abstract - ***Ultra*SPARC's IEEE-754 compliant floating point divide and square root implementation is presented. Three overlapping stages of SRT radix-2 quotient selection logic enable an effective radix-8 calculation at 167 MHz while only a single radix-2 quotient selection logic delay is seen in the critical path. Speculative partial remainder and quotient calculation in the main datapath also improves cycle time. The quotient selection logic is slightly modified to prevent the formation of a negative partial remainder for exact results. This saves latency and hardware as the partial remainder no longer needs to be restored before calculating the sticky bit for rounding.**

I. Introduction

The SRT algorithm provides a means of performing non-restoring division [1], [2]. More bits of quotient are developed per iteration with higher radices, but at a cost of greater complexity. A simple SRT radix-2 floating point implementation (Fig. 1) requires that the divisor and dividend both be positive and normalized, $1/2 \le D, Dividend < 1$. The initial shifted partial remainder, *2PR[0]*, is the dividend. Future partial remainders are developed according to the following equation,

$$PR_{i+1} = 2PR_i - q_{i+1}D. \qquad (1)$$

where q is the quotient digit {-1, 0, or +1} which is solely determined by the value of the previous partial remainder and is independent of the divisor, an attractive feature for square root. Discussion of the quotient digit selection function will be deferred until the next section. The partial remainder is often kept in redundant form so that carry-save adders can be used instead of slower and larger carry-propagate adders. The partial remainder is converted to non-redundant form after the desired precision is reached. The quotient digits can also be kept in redundant form and converted to non-redundant form at the end, or the quotient and quotient minus one (*Q* and *Q-1*) can be generated on the fly according to rules developed by Ercegovac [3].

The SRT algorithm can also be used for square root allowing utilization of existing division hardware. The simplified square root equation looks surprisingly similar to that of division [4]:

$$PR_{i+1} = 2PR_i - q_{i+1}(2Q_i + q_{i+1}2^{-(i+1)}). \qquad (2)$$

The terms in parentheses are the effective divisor. For square root, the so-called divisor is a function of the previous quotient bits (root bits to be more precise) [4], hence on-the-fly quotient generation is required.

Quotient selection logic (qslc) for a radix-2 SRT implementation will be discussed in Sections II and III with emphasis on how it can be modified to better constrain the partial remainder in the case of exact results. Preventing the partial remainder from unnecessarily going negative for exact results leads to a one cycle savings in generating the sticky bit.

It will be demonstrated in Section IV that overlapping radix-2 quotient selection logic stages can provide an effective timing solution for generating multiple bits of quotient per cycle. The timing benefits of speculative datapath calculations of the partial remainder, quotient, and next divisor occurring in conjunction with quotient digit selection will be shown. A comparison of overlapped radix-2 versus radix-4 implementations will also be discussed.

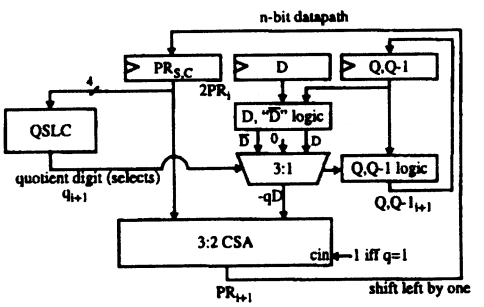


Fig. 1. Simple SRT Radix-2 Divide, Square Root Implementation

II. Quotient Selection Logic

The logic which generates quotient selection digits is the central element of an SRT division implementation.

Reprinted from *Proceedings of the 12th Symposium on Computer Arithmetic*, pp. 155-162, July 1995.

Early research indicated that only the upper three bits of the redundant partial remainder are necessary inputs for a radix-2 quotient digit selection function [5], [6]. However, more recent studies have shown that four bits are required to correctly generate quotient digit selection digits and keep the partial remainder within prescribed bounds [7], [8], [9], [10]. The selection rules can be expressed as:

$$q_{i+1} = \begin{cases} 1, \text{if } 0 \le 2PR_i \le 3/2 \\ 0, \text{if } 2PR_i = -1/2 \\ -1, \text{if } -5/2 \le 2PR_i \le -1 \end{cases} \quad (3)$$

The quotient selection logic is designed to guess correctly or overestimate the true quotient result. e.g. predicting 1 instead of 0, or 0 instead of -1. The SRT algorithm corrects itself later if the wrong quotient digit has been chosen.

TABLE I
TRUTH TABLE FOR RADIX-2 QUOTIENT SELECTION LOGIC

$2PR_{i,estimated}$	quotient $digit_{i+1}$	comments
100.0	don't care	2PR never < -5/2
100.1	don't care	2PR never < -5/2
101.0	-1*	2PR never < -5/2, but 2PR could be 101.1 when $2PR_{est}$ is 101.0
101.1	-1	
110.0	-1	
110.1	-1	
111.0	-1	2PR could = 111.1
111.1	0	2PR could = 000.0
000.0	+1	
000.1	+1	
001.0	+1	
001.1	+1	
010.0	don't care	2PR never > 3/2
010.1	don't care	2PR never > 3/2
011.0	don't care	2PR never > 3/2
011.1	don't care	2PR never > 3/2

The truth table for SRT radix-2 quotient selection logic has several don't care inputs because the partial remainder is constrained $-5/2 \le 2PR_i \le 3/2$. The estimated partial remainder is always less than or equal to the true partial remainder because the lower bits are ignored. There is a single case, $2PR_{est} = 101.0$, where the estimated partial remainder can appear to be out of bounds. By construction, the real partial remainder is within the negative bound, so -1 is the appropriate quotient digit to select. There are two other cases where the quotient digit selected based on the estimated partial remainder differs from what would be chosen based on the real partial remainder. However, in both of these instances of "incorrect" quotient digit selection, the quotient is not underestimated and the partial remainder is kept within prescribed bounds, so the final result will still be generated correctly.

For increased testability, most designs today are scannable. All registers are stitched together in a chain to allow a special test mode known as scan. During scan, external values can be optionally be shifted into these registers, the system can be clocked for one or more cycles, and the new register values can be shifted out and observed. These capabilities aid in functional and timing debug.

While loading the scan chain, the partial remainder flip flops can take on any value. Logic simplification for don't care cases should ensure that a unique quotient digit is always selected (i.e. the quotient digits selects are 1-hot) for all input combinations to prevent contention of multiplexer selects. The simplified truth table follows.

TABLE II
SIMPLIFIED QSLC TRUTH TABLE

$2PR_{i,estimated}$	quotient $digit_{i+1}$
0xx.x	+1
111.1	0
1xx.x	-1

III. STICKY BIT CALCULATION

Floating point operations generate a sticky bit along with the result in order to indicate whether the result is inexact. The sticky bit is also used with the guard and round bits for rounding according to IEEE Standard 754 [11]. For divide and square root operations, the sticky bit is determined by checking if the final partial remainder is non-zero. The final partial remainder is defined as the partial remainder after the desired number of quotients bits have been calculated. Since the partial remainder is in redundant form, a carry-propagate addition is performed prior to zero-detection (See Fig. 3a).

A. Exact Results

At first glance, the above solution seems perfectly reasonable for all final partial remainder possibilities, positive or negative. However, in the rare case where the result is exact, the final partial remainder will be equal to the negative divisor. For example, consider a number divided by itself (Fig.2). Since the dividend is positive and normalized, the quotient digit from the first iteration is one. For the next iteration, the partial remainder is zero which causes the second quotient digit to be one. For all subsequent iterations, the partial remainder will equal the negative divisor and quotient digits of minus one will be selected. After the last iteration, performing a sign detect on the final partial remainder indicates that *Q-1* should be chosen which is in fact the correct result. However, this same final partial remainder is non-zero which erroneously suggests an inexact result.

PR[0] = init dividend/2 = D/2	q[1] =+1	Q=1	Q-1=0
PR[1] = 2(D/2) - (1)D = 0	q[2] =+1	Q=11	Q-1=10
PR[2] = 2(0) - (1)D = - D	q[3] =- 1	Q=101	Q-1=100
PR[3] = 2(-D) - (-1)D = - D	q[4] =- 1	Q=1001	Q-1=1000
PR[n] = 2(-D) - (-1)D = - D	q[n+1] =- 1	Q=100..001	Q-1=100..000

Fig. 2. Divide iterations for a number divided by itself.

This problem extends to any division operation for which the result should be exact because the quotient selection logic is defined to guess positive for a zero partial remainder and correct for it later. The simplest solution is to restore negative final partial remainders by adding the divisor before performing zero-detection. Given the area expense of an additional carry-propagate adder, the solution should try to take advantage of existing hardware. Two ways to achieve this are shown in Figs. 3b and 3c.

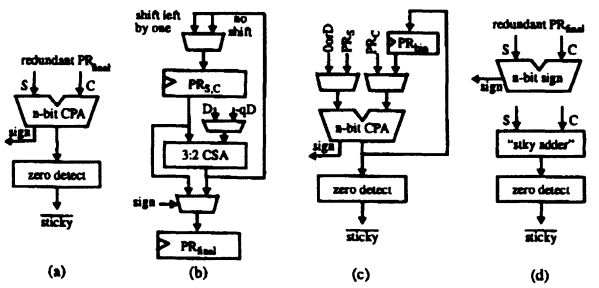


Fig. 3. Zero-detection and sign-detection on the final partial remainder.

Option 1 (Fig. 3b) takes advantage of existing csa hardware for restoration while option 2 (Fig. 3c) re-uses the carry-propagate adder. Both alternatives add extra multiplexer hardware and require the sticky bit calculation to take an additional cycle when the preliminary final partial remainder is negative. The first option especially impacts cycle time for basic iterations since the multiplexer is on the partial remainder formation critical path. Variable latency instructions in a pipelined superscalar processor make instruction scheduling and bypass control logic much more complex and is generally undesirable. Thus the net effect of restoring negative partial remainders is to add another cycle of latency for all divide and square root operations.

B. Improved Quotient Selection Algorithm

Enhancing the quotient digit selection function to prevent formation of a negative partial remainder for exact results is an ideal solution because it saves hardware and improves latency. This can be achieved by choosing a quotient digit of zero instead of one when the partial remainder is zero. This suggests choosing q=0 for $2PR_{est}$=000.0. Since the quotient digit selection function works on an estimated partial remainder, caution is required. An estimated partial remainder (shifted) could appear to be less than 1/2 when, in reality, adding the lower bits causes a 1 to propagate into the lowermost bit of the upper four bit partial remainder.

If the full partial remainder is 1/2 or above, q=1 should always be chosen since the divisor is constrained $1/2 \le D < 1$. The true quotient bit could be one. It will be corrected later if the divisor was greater than the partial remainder. There is no way to correct for under-estimation if q=0 is selected when q=1 was the correct quotient digit. The result will be irreversibly incorrect and the next partial remainder will be out of bounds.

Performing binary addition on the full partial remainder eliminates the estimation problem, but defeats the timing benefits of SRT division. q=0 could be chosen only when the full partial remainder is zero, but such detection would also be detrimental to timing.

A simple alternative is to detect a possible carry propagation into the least significant bit of the partial remainder. This can be done by looking at the fifth most significant bits of the redundant partial remainder, $PR_{S,msb-4}$ and $PR_{C,msb-4}$. If they are both zero, then propagation is not possible, and q=0 should be chosen; otherwise q=1 should be chosen. Even though lower bits of the partial remainder could be non-zero, the partial remainder is still within prescribed bounds and the correct result will be generated. As far as timing is concerned, the carry-propagate addition is still performed on four bits.

TABLE III
REFINED QSLC TRUTH TABLE

$2PR_{i,estimated}$	quotient $digit_{i+1}$
000.0 ($2PR_{S,C,msb-4}$ both 0)	0
0xx.x	+1
111.1	0
1xx.x	-1

Fig. 4 shows a logic implementation of this enhanced quotient digit selection function. In practice, the four bit adder and subsequent logic are merged into five stages of logic for more efficient timing and area utilization.

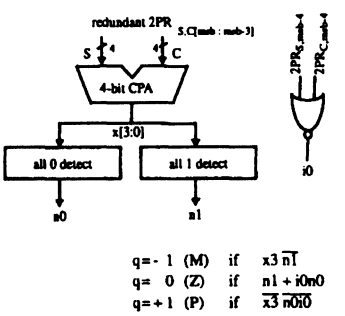


Fig. 4. Simple implementation of modified QSLC.

The number of additional gates needed to implement the new radix-2 quotient digit selection logic is relatively small. From Spice analysis, the impact on the qslc critical timing path was under five percent. There is an implementation dependent timing trade-off between slower quotient selection logic and eliminating the partial remainder restoration cycle at then end. Notice that if slowing down qslc doesn't limit the processor cycle time, there will always be a performance gain from saving one cycle of latency.

C. Parallel Sign Calculation and Zero-detection

It is possible to save hardware while also performing sign detection and sticky bit calculation in parallel as shown in Fig. 3d. Instead of using a full 59 bit adder, a 59 bit sign detect adder can be used, slightly improving timing, but mainly saving area. Zero-detection can be done without an explicit addition to convert the redundant partial remainder into binary.

$$t_i = \left(s_i \oplus c_i\right) \oplus \left(s_{i-1} + c_{i-1}\right) \tag{4}$$

where s_i and c_i are the sum and carry values of the final partial remainder. The sticky bit is computed by:

$$sticky = t_0 + t_1 + \ldots + t_n \tag{5}$$

This method generates inputs to the zero-detector with a 3-input xor delay instead of the delay of a 59 bit carry-propagate adder, a significant net savings.

IV. Overlapping Radix-2 Stages

The biggest performance gain is achieved by maximizing the number of iterations per cycle. As the number of result bits formed per cycle increases, the relative importance of saving one cycle of latency, as described in Section III, grows. A straightforward way of generating n result bits per cycle is to serialize the basic SRT radix-2 implementation. This solution is not attractive since the critical path includes n quotient selection logic delays and n carry save adder delays.

A. Optimal Timing via QSLC Overlapping

Overlapping quotient selection logic for the first and second iterations [12] as shown in Fig. 5 yields better timing results since only one qslc is in the critical path. Overlapping is achieved by performing $+D$ and $-D$ operations on the upper bits of the partial remainder while the first quotient selection bit is being determined. In this way, the second quotient digit selection can start before the first is finished.

Maximal overlapping can be extended to three bits per cycle to have an effective radix-8 implementation.

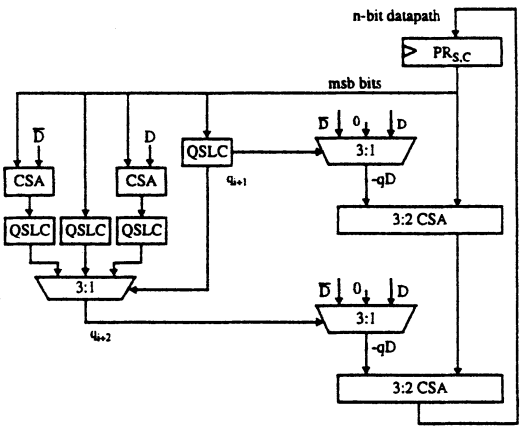


Fig. 5. Two overlapped SRT radix-2 QSLC stages.

Analyzing the possible partial remainders needed for the third quotient digit selection, as depicted in Fig. 6, shows that only seven csa's and qslc's are needed rather than nine. Since quotient selection logic is area intensive, a 22% reduction is quite beneficial.

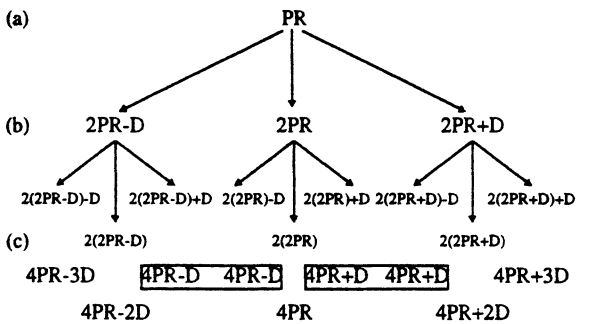


Fig. 6. Possible partial remainder values after the first and second iterations. (a) initial PR_i, (b) three possible PR_{i+1}, (c) seven possible PR_{i+2}.

Further timing improvement is also possible. By speculatively calculating the next partial remainder, quotient, and divisors for each possible quotient selection digit, the delay of a datapath carry-save adder can be masked by the longer qslc operation occurring in parallel. Fig. 7 shows the overall *Ultra*SPARC floating point divide, square root implementation. The critical path is reduced to 1 qslc, 2 csa's, and 3 muxes.

There is a timing, area trade-off. Overlapping improves timing at a cost of additional speculative hardware. The focus of this study is to optimize timing to the utmost with area minimization as a secondary goal.

B. Extension

In theory, n qslc stages can be overlapped. Assuming speculative datapath partial remainder calculations each iteration, the critical path for n bits per cycle is 1 qslc, (n-1) csa's, and n muxes. The incremental timing cost is one csa and one mux. There are 2^n-1 partial remainder

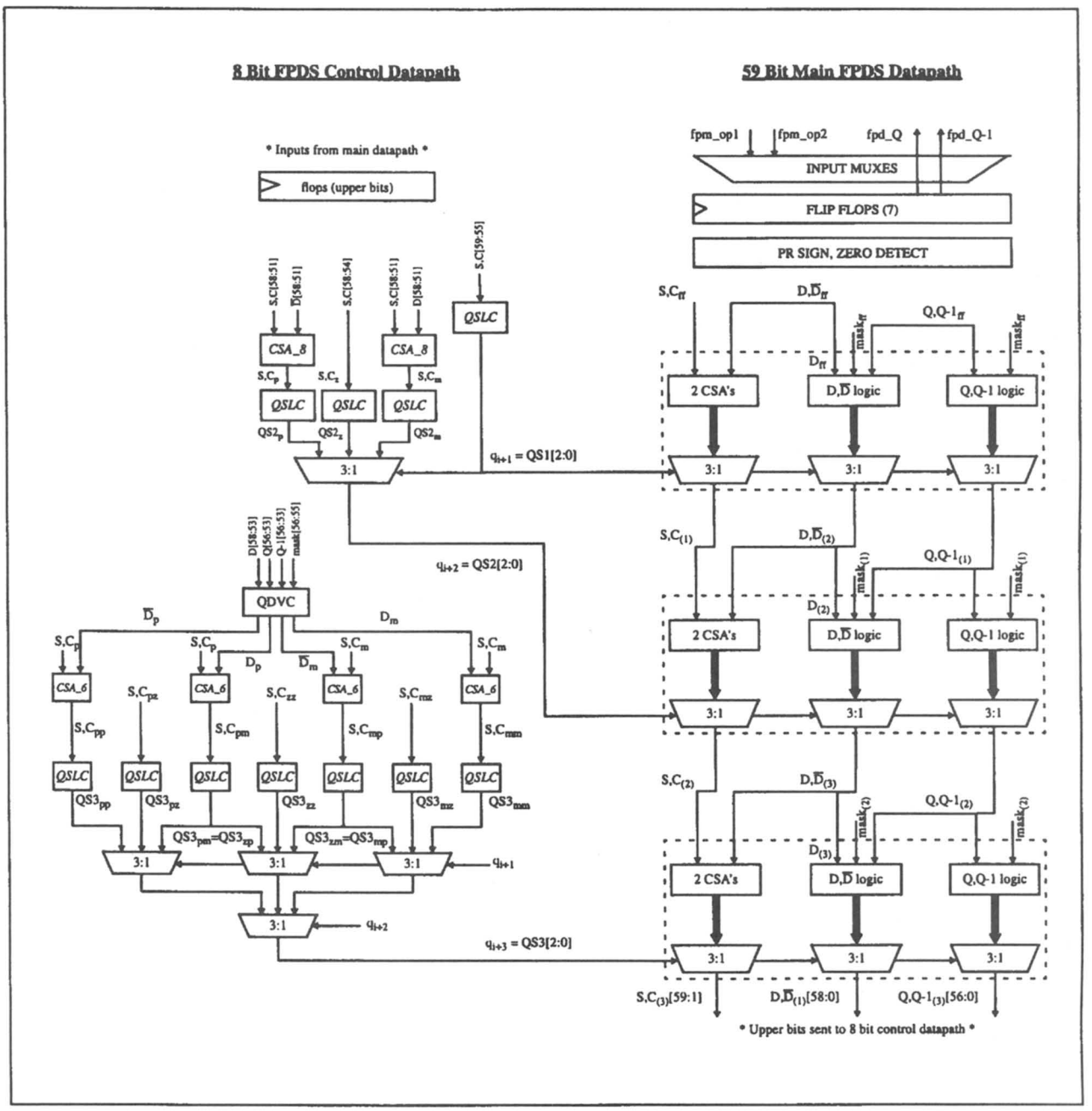


Fig. 7. *Ultra*SPARC radix-8 floating point divide, square root implementation with three overlapped radix-2 stages and speculative datapath calculation.

possibilities for the *n*th qslc stage. They are in the range:

$$2^{n-1}PR_i + [-(2^{n-1}-1)D,..,0,..,(2^{n-1}-1)D]. \quad (6)$$

Therefore, the incremental qslc cost for the *n*th overlapped stage is 2^n-1 rather than 3^{n-1} as suggested by Taylor [12]. In practice, overlapping two to four stages makes the most sense. As *n* gets higher, the number of qslc's grows exponentially making the area cost prohibitive. In addition, greater fanout leads to increased wire and gate loads which significantly lessen the timing benefits. Table IV summarizes timing and qslc cost considerations for overlapped radix-2 stages.

TABLE IV
PERFORMANCE, COST TABLE FOR MAXIMUM OVERLAPPING

stages	critical path	tot qslc's	delta critical path	delta qslc's
1	qs + mux	1	-	-
2	qs + pr + 2mux	4	pr + mux	3
3	qs + 2pr + 3mux	11	pr + mux	7
4	qs + 3pr + 4mux	26	pr + mux	15
n	qs + (n-1)pr +(n)mux	$\sum_{i=1}^{n} 2^i - 1$	pr + mux	$2^n - 1$

There is a limit on how much overlapping is necessary to achieve the optimal radix-2 implementation critical path as illustrated in a four bit per cycle example. Suppose the quotient selection logic delay is three times a carry-save adder delay. Then the quotient digits from the first stage qslc will be ready at the same time as the third stage partial remainder bits are entering the fourth stage qslc's. The first level of three-to-one muxes following the fourth stage of qslc's can be moved before the quotient selection logic as shown in Fig. 8. Thus, the fourth stage number of qslc's is reduced from fifteen to seven while achieving the same timing as with maximum overlapping. In general, the optimal degree of overlapping will depend on the relative csa and qslc delays, and need only be sufficient to mask the delay of previous quotient selection logic stages.

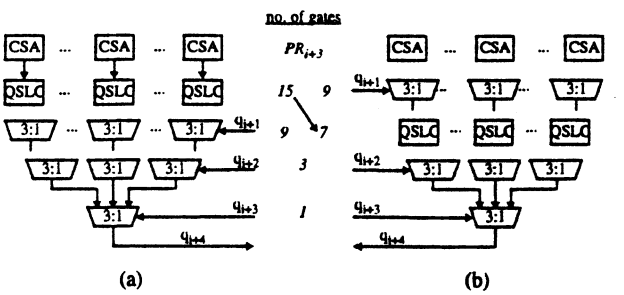


Fig. 8. Optimal radix-2 timing with (a) maximum overlapping, (b) reduced overlapping.

C. Overlapped Radix-2 versus Radix-4

Overlapping radix-2 stages yields better timing results than overlapping radix-4 stages. With radix-4, the upper eight bits of the redundant partial remainder and upper four bits of the divisor are sent to the quotient selection logic. The logic cannot be merged as with radix-2, so there is an explicit eight bit binary addition followed by a ten input, forty-four product term PLA [12].

Comparison of the critical path gate delays (gd) for two bits per cycle and four bits per cycle confirms better timing from a radix-2 based solution. The critical paths for two bits per cycle are as follows.

$$\textit{radix-2:}\quad \underset{5}{qslc} + \underset{1.8}{csa} + \underset{(2\,x\,1)}{2\,mux3.} \qquad (8.8gd) \quad (7)$$

$$\textit{radix-4:}\quad \underset{6}{add8} + pla + \underset{1}{mux5.} \qquad (7gd+pla) \quad (8)$$

Two overlapped radix-4 stages are implemented in the same way as depicted for radix-2 in Fig. 5. The critical paths for four bits per cycle are shown below.

$$\textit{radix-2:}\quad \underset{5}{qslc} + \underset{(3x1.8)}{3\,csa} + \underset{(4\,x\,1)}{4\,mux3.} \qquad (14.4gd) \quad (9)$$

$$\textit{radix-4:}\quad \underset{1.8}{csa} + \underset{6}{add8} + pla + \underset{(2\,x\,1)}{2\,mux5.} \qquad (10)$$

$$= (9.8gd+pla)$$

A 10-input, 44-product term pla takes more than five gate delays, so for both two bits per cycle and four bits per cycle, overlapping radix-2 stages produces better timing. This analysis implies that a combined radix-4, radix-2 approach to yield an effective radix-8 solution is not faster than simply overlapping three radix-2 stages.

V. PROCESSOR IMPLEMENTATION

*Ultra*SPARC performs non-blocking divide and square root operations. Generating three result bits per cycle at 167 Mhz, the latency is twelve cycles for single precision (SP) and twenty-two cycles for double precision (DP). The divide and square root unit contains seventy-thousand transistors implemented in 0.5um CMOS technology (Fig. 9).

Instructions are both issued and completed through the pipelined floating point multiplier which formats operands, calculates the exponent, and performs rounding. The floating point multiplier and divider share the same register file read ports for operands, so divide and multiply instructions are never issued at the same time. Therefore the multiplier is always available for operand formatting and exponent calculation when a divide instruction is issued. Multiply instructions can be issued during subsequent cycles while the divider is iterating. Four cycles prior to division completing, a

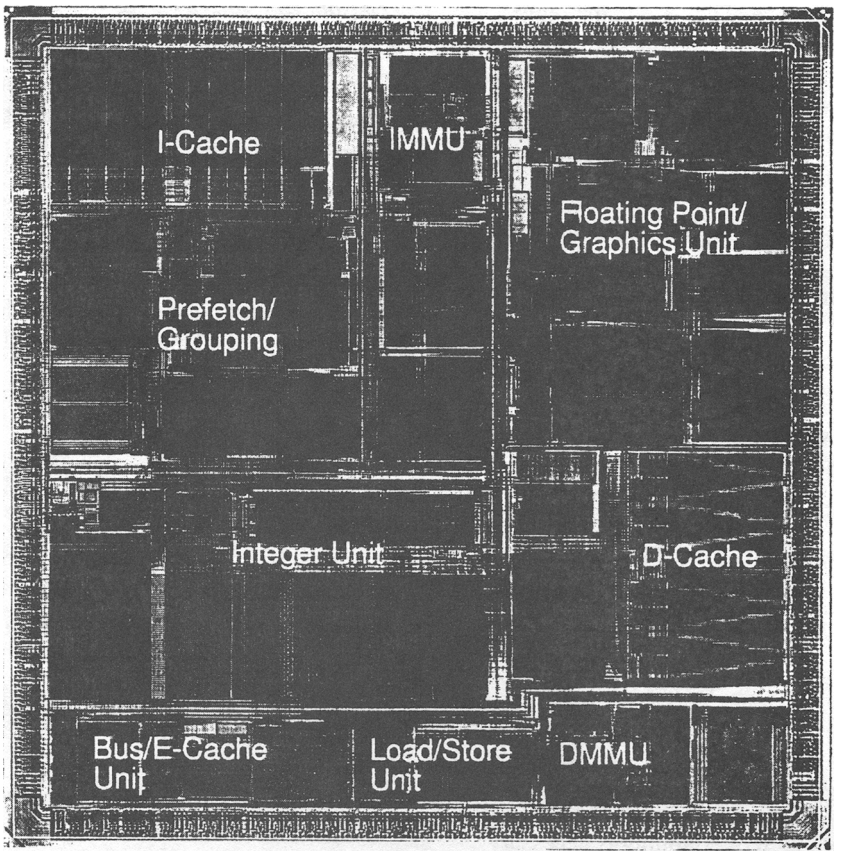
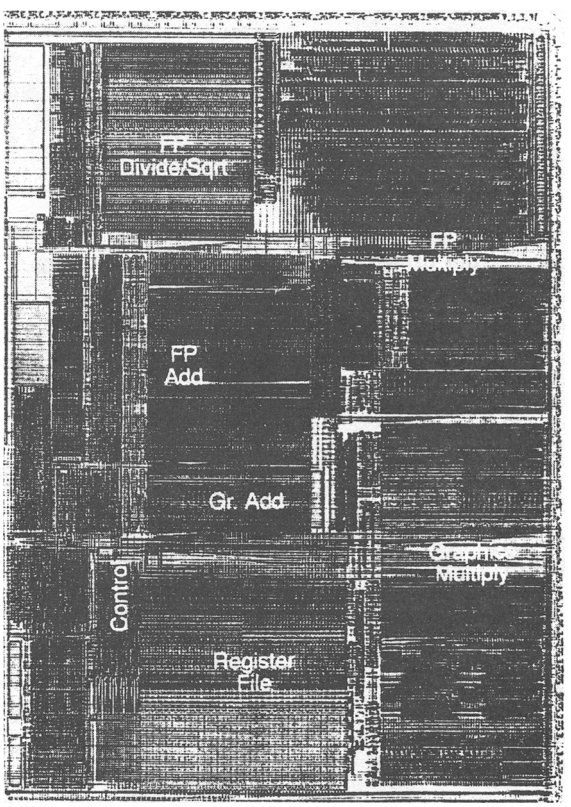


Fig. 9. *Ultra*SPARC (above), *Ultra*SPARC floating point, graphics unit (below).



multiply slot is reserved so that the divider can re-use the final stage multiplier rounding hardware. Since *Ultra*SPARC performs in-order execution, there will be a one cycle delay only in the unlikely event that the superscalar processor has been able to issue independent instructions for 8 cycles (SP) or 18 cycles (DP) and an independent multiply instruction is ready to be issued during this exact cycle. Thus making use of the multiplier yields significant hardware savings with near zero performance impact.

The modified quotient selection logic algorithm slowed the internal floating point divider critical path by approximately 100ps, or by less than two percent. The limiting timing path on the processor was slower, so there was no negative impact from the improved qslc. The full benefit of eliminating the cycle for restoring the partial remainder prior to sticky detection was realized.

Division and square root operations were thoroughly tested with 100% toggle coverage and 100% finite state machine arc coverage in a stand-alone test (SAT) environment, at the cpu level, and in silicon. Over 860,000 directed vectors and 140,000 random vectors were applied at the SAT level. Included were pseudo-exhaustive double precision tests for division in which all combinations of the upper eight bits of the dividend and divisor were sequenced (64K vectors). The same was done for square root in which all combinations of the upper thirteen bits of square root operands for odd and even exponents were sequenced (16K vectors).

A radix-2 solution was preferable over radix-4 for a number of reasons including timing, as described earlier, and greater flexibility. From the outset, it was known that it might be necessary to scale back the number of bits per cycle due to timing or area considerations. That raised the specter of having to reduce to two bits per cycle in a radix-4 implementation. Going with radix-2 provided a better safety option since three bits per cycle yields 6% better overall floating point performance (SPECfp92) than two bits per cycle [13].

Implementing square root did not come for free. While it is true that square root completely re-uses existing divide hardware, additional logic was required to generate the so-called divisors used in square root. In addition, the quotient had to be developed on the fly, as opposed to using a shift register and assimilating the redundant bits during the final cycle, since it was needed for the divisor calculation. Additional datapath feedthrough tracks were also necessary. An estimated 15% of the area went to support square root. With careful design, the logical critical path for square root was made the same as for divide. The additional area required to support square root, though, did impact timing since wire delays were greater.

VI. Conclusion

High-speed floating point division and square root is achieved by overlapping radix-2 quotient selection logic stages and speculatively calculating the partial remainder, quotient, and next divisor. An enhanced quotient digit selection function prevents the working partial remainder from becoming negative if the result is exact. This translates into a one cycle savings since negative partial remainders no longer need to be restored before calculating the sticky bit.

Acknowledgment

The authors would like to thank Marc Tremblay and Guy Steele for reviewing the preliminary draft, and Nasima Parveen and Richard Landes for their contributions to verification and physical design.

References

[1] J. E. Robertson, "A new class of digital division methods," *IEEE Trans. Comput.*, vol. C-7, pp. 218-222, Sept. 1958.

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

[3] M. D. Ercegovac and T. Lang, "On-the-fly rounding," *IEEE Trans. Comput.*, vol. 41, no. 12, pp. 1497-1503, Dec. 1992.

[4] M. D. Ercegovac and T. Lang, "Radix-4 square root without initial PLA," *IEEE Trans. Comput.*, vol. 39, no. 8, pp. 1016-1024, Aug. 1990.

[5] S. Majerski, "Square root algorithms for high-speed digital circuits," *Proc. Sixth IEEE Symp. Comput. Arithmetic*. pp. 99-102, 1983.

[6] D. Zuras and W. McAllister, "Balanced delay trees and combinatorial division in VLSI," *IEEE J. Solid-State Circuits*, vol. SC-21, no. 5, pp. 814-819, Oct. 1986.

[7] M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994, ch 3.

[8] S. Majerski, "Square-rooting algorithms for high-speed digital circuits," *IEEE Trans. Comput.*, vol. C-34, no. 8, pp. 724-733, Aug. 1985.

[9] P. Montuschi and L. Ciminiera, "Simple radix 2 division and square root with skipping of some addition Steps," *Proc. Tenth IEEE Symp. Comput. Arithmetic*. pp. 202-209, 1991.

[10] V. Peng, S. Samudrala, and M. Gavrielov, "On the implementation of shifters, multipliers, and dividers in floating point units," *Proc. Eighth IEEE Symp. Comput. Arithmetic*. pp. 95-101, 1987.

[11] "IEEE standard for binary floating-point arithmetic," ANSI/IEEE Standard 754-1985, New York, The Institute of Electrical and Electronic Engineers, Inc., 1985.

[12] G. S. Taylor, "Radix 16 SRT dividers with overlapped quotient selection stages," *Proc. Seventh IEEE Symp. Comput. Arithmetic*. pp. 95-101, 1985.

[13] M. Tremblay, "A fast and flexible performance simulator for micro-architecture trade-off analysis on UltraSPARC," Submitted to the 1995 *Design Automation Conference*.

Radix-4 Square Root Without Initial PLA

MILOŠ D. ERCEGOVAC, MEMBER, IEEE, AND TOMAS LANG

Abstract—**A systematic derivation of a radix-4 square-root algorithm using redundant residual and result is presented. Unlike other similar schemes it does not use a table lookup or PLA for the initial step, resulting in a simpler implementation without any time penalty. The scheme can be integrated with division and also incorporates an on-the-fly conversion and rounding of the result, thus eliminating a carry-propagate step to obtain the final result. The result-digit selection uses 3 bits of the result and 7 bits of the estimate of the residual.**

Index Terms—**Digital arithmetic, digit-recurrence, on-the-fly conversion, radix-4 square root, redundant representation.**

I. Introduction

SEVERAL implementations of radix-4 square root have been presented in the literature [9], [6], [5], [10], [8]. In all these cases, a table lookup or a special PLA is included for the determination of the first few bits of the result, whereas another PLA implements the result-digit selection for the remaining radix-4 digits. In this paper, we present a systematic derivation of the algorithm and show that this initial PLA is not necessary; this results in a simpler implementation without any penalty in execution time. As in the other designs, the implementation can be combined with division: the result-digit selection and the recurrence form are identical in all steps.

To obtain a fast implementation, as done in [6], [5], [10], and [8], redundant addition is used and the result-digit selection depends on low-precision estimates of the residual and of the partial result. This requires that the result digit be from a redundant digit-set. As the other referenced implementations, we use the set $\{-2, -1, 0, 1, 2\}$ to simplify the multiple formation required by the recurrence.

Two alternative redundant adders are possible: carry-save adder or signed-digit adder, each having advantages and drawbacks. The advantage of using a carry-save adder is that the adder slice is just a full adder, whereas the signed-digit adder is more complex and slower [7]. On the other hand, since the result is produced in a signed-digit representation and the partial result is one input to the adder, if carry-save addition is used this partial result has to be converted to conventional representation, whereas it can be used directly for signed-digit addition. We choose the carry-save alternative because the result has to be converted anyhow to conventional representation and to do this we use an on-the-fly converter, which after each iteration produces the partial result in conventional form. Moreover, during this conversion on-the-fly rounding is performed [3].

We describe a sequential implementation, in which the hardware of one iteration step is reused for each of the digits of the result. Of course, it is possible to have a combinational implementation, in which the iteration step is replicated. The selection between these alternatives is influenced by cost, speed, and throughput considerations.

The operation is defined by

$$s = x^{1/2} - \epsilon$$

where x is the operand, s is the result, and ϵ is the error. The algorithm is presented for *fractional* operand x and result s. For floating-point representation and normalized operand, it is necessary to scale the operand to have an even exponent (to allow the computation of the exponent). Consequently,

$$1/4 \le x < 1 \quad 1/2 \le s < 1.$$

If s has m fractional radix-4 digits then for a correct result the error is bounded so that

$$|\epsilon| < 4^{-m}.$$

In Section II, we describe the algorithm and in Sections III and IV we discuss its implementation.

II. Recurrence and Square Root Step

The algorithm is based on a continued-sum recurrence. We now develop the digit-recurrence and show the implementation of the corresponding iteration step.

A. Recurrence and Bound

Each iteration of the recurrence produces one digit of the result, most significant digit first. Let us call $S[j]$ the value of the partial result after j iterations, that is

$$S[j] = \sum_{i=0}^{j} s_i 4^{-i} \qquad s_i \in \{-2, -1, 0, 1, 2\} \qquad (1)$$

(the digit s_0 is needed to represent a result value $s \ge 2/3$, since the representation of s is in signed-digit form and the maximum value of s_i is 2).

The final result is then

$$s = S[m] = \sum_{i=0}^{m} s_i r^{-i}$$

Manuscript received October 26, 1989; revised March 6, 1990. This work was supported in part by NSF Grant MIP-8813340, Composite Operations Using On-Line Arithmetic for Application-Specific Parallel Architecture: Algorithms, Design, and Experimental Studies.

The authors are with the Department of Computer Science, School of Engineering and Applied Science, University of California, Los Angeles, CA 90024.

IEEE Log Number 9036137.

Reprinted from *IEEE Transactions on Computers*, Vol. 39, No. 8, pp. 1016-1024, August 1990.

and the result has to be correct for m-digit precision, that is,

$$|x^{1/2} - s| < 4^{-m}. \tag{2}$$

We define an error function ϵ so that its value after j steps is

$$\epsilon[j] = x^{1/2} - S[j]. \tag{3}$$

To have a correct final result this error has to be bounded. Since

$$s = S[j] + \sum_{i=j+1}^{m} s_i 4^{-i}$$

we get from (2) and (3)

$$\min\left(\sum_{i=j+1}^{m} s_i 4^{-i}\right) - 4^{-m} < \epsilon[j] < \max\left(\sum_{i=j+1}^{m} s_i 4^{-i}\right) + 4^{-m}.$$

Since the minimum (maximum) digit value is -2 (2), we get

$$-\frac{2}{3}4^{-j} \le \epsilon[j] \le \frac{2}{3}4^{-j}. \tag{4}$$

Introducing (3) in (4) and transforming to eliminate the square root operation (add $S[j]$ and obtain the square), we get

$$\frac{4}{9}4^{-2j} - \frac{4}{3}4^{-j}S[j] + S[j]^2 \le x \le \frac{4}{9}4^{-2j} + \frac{4}{3}4^{-j}S[j] + S[j]^2.$$

Subtracting $S[j]^2$ we obtain

$$\frac{4}{9}4^{-2j} - \frac{4}{3}4^{-j}S[j] \le x - S[j]^2 \le \frac{4}{9}4^{-2j} + \frac{4}{3}4^{-j}S[j]. \tag{5}$$

That is, we have to compute $S[j]$ such that $x - S[j]^2$ is bounded according to (5). We now define a residual (or partial remainder) w so that

$$w[j] = 4^j(x - S[j]^2). \tag{6}$$

From (5) the bound on the residual is

$$-\frac{4}{3}S[j] + \frac{4}{9}4^{-j} \le w[j] \le \frac{4}{3}S[j] + \frac{4}{9}4^{-j}. \tag{7}$$

Since from (1) $S[-1] = 0$, we get from (6) the initial condition

$$w[-1] = 4^{-1}x. \tag{8}$$

From (6) and (1) the basic recurrence of the square root algorithm is

$$w[j+1] = 4w[j] - 2S[j]s_{j+1} - s_{j+1}^2 4^{-(j+1)}. \tag{9}$$

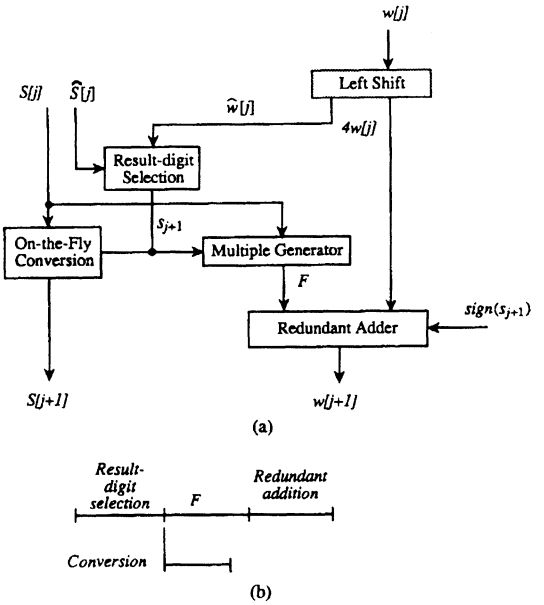


Fig. 1. (a) Square root step. (b) Timing.

B. Implementation of Square Root Step

The square root algorithm consists in performing m iterations of the recurrence (9). Moreover, each iteration consists of four subcomputations [Fig. 1(a)].

1) One digit arithmetic left-shift of $w[j]$ to produce $4w[j]$.

2) Determination of the result digit s_{j+1} using a result-digit selection function *Select*. The value of the digit s_{j+1} is selected so that the application of the recurrence produces a $w[j+1]$ that satisfies the bound (7). The function has as arguments $\hat{w}[j]$ (an estimate of $4w[j]$) and $\hat{S}[j]$ (an estimate of $S[j]$) and produces s_{j+1}. That is,

$$s_{j+1} = \mathit{Select}(\hat{w}[j], \hat{S}[j]).$$

3) Formation of

$$F = -2S[j]s_{j+1} - s_{j+1}^2 4^{-(j+1)}. \tag{10}$$

4) Addition of F and $4w[j]$ to produce $w[j+1]$.

The four subcomputations are executed in sequence as indicated in the timing diagram of Fig. 1(b). Note that no time has been allocated for the arithmetic shift since it is performed just by suitable wiring. Moreover, the relative magnitudes of the delay of each of the components depend on the specific implementation.

To have a fast recurrence step we use a carry-save adder (a signed-digit adder could also be used) and a result-digit selection that depends on low-precision estimates of the residual and of the partial result. To achieve this, it is necessary to have a redundant representation of the result digit. As indicated before, we use the symmetric digit-set

$$s_i \in \{-2, -1, 0, 1, 2\} \tag{11}$$

because it allows a simpler implementation of the adder input F (no multiple of three is required). Moreover, the signed

result-digit makes it necessary to use an on-the-fly conversion to produce $S[j]$ in a conventional form for the formation of F.

Section III presents the result-digit selection and Section IV discusses the formation of F.

III. Result-Digit Selection Function

The selection function determines the value of the result digit s_{j+1} as a function of an estimate of the residual $w[j]$ and of an estimate of the partial result $S[j]$. Two fundamental conditions must be satisfied by a result-digit selection: *containment* and *continuity*. These conditions determine a *selection interval* for each value of s_{j+1}. We now develop these selection intervals.

A. Containment Condition and Selection Intervals

Let the bounds of the residual $w[j]$ be called $\underline{B}$ and $\bar{B}$, that is,

$$\underline{B}[j] \le w[j] \le \bar{B}[j]. \tag{12}$$

Define the *selection interval* of $4w[j]$ for $s_{j+1} = k$ to be $[L_k, U_k]$. That is, L_k (U_k) is the smallest (largest) value of $4w[j]$ for which it is *possible* to choose $s_{j+1} = k$ and keep $w[j+1]$ bounded. Therefore, from the recurrence

$$L_k[j] \le 4w[j] \le U_k[j] \rightarrow \underline{B}[j+1] \le 4w[j] - 2S[j]k - k^2 4^{-(j+1)} \le \bar{B}[j+1]. \tag{13}$$

Consequently,

$$U_k[j] = \bar{B}[j+1] + 2S[j]k + k^2 4^{-(j+1)}$$
$$L_k[j] = \underline{B}[j+1] + 2S[j]k + k^2 4^{-(j+1)}. \tag{14}$$

Since $\bar{B}[j]$ and $\underline{B}[j]$ are the upper bound of the interval for $k = 2$ and the lower bound for $k = -2$, respectively, we get

$$U_2[j] = 4\bar{B}[j] \quad L_{-2}[j] = 4\underline{B}[j].$$

Introducing these values in (14) we get

$$\bar{B}[j+1] + 2S[j] \times 2 + 2^2 4^{-(j+1)} = 4\bar{B}[j]$$
$$\underline{B}[j+1] - 2S[j] \times 2 + 2^2 4^{-(j+1)} = 4\underline{B}[j]. \tag{15}$$

This results in

$$\bar{B}[j] = \frac{4}{3} S[j] + \frac{4}{9} 4^{-j}$$
$$\underline{B}[j] = -\frac{4}{3} S[j] + \frac{4}{9} 4^{-j}. \tag{16}$$

To show that (16) is correct it is sufficient to replace in (15). Note that these bounds are identical to those obtained in (7). If this were not the case, we would use the tighter bounds.

From (14) and (16), the selection intervals are given by the expressions

$$U_k[j] = \frac{4}{3} S[j+1] + \frac{4}{9} 4^{-(j+1)} + 2S[j]k + k^2 4^{-(j+1)}$$
$$L_k[j] = -\frac{4}{3} S[j+1] + \frac{4}{9} 4^{-(j+1)} + 2S[j]k + k^2 4^{-(j+1)}.$$

Since $S[j+1] = S[j] + k \times 4^{-(j+1)}$ we get

$$U_k[j] = 2S[j]\left(k + \frac{2}{3}\right) + \left(k + \frac{2}{3}\right)^2 4^{-(j+1)} \tag{17a}$$
$$L_k[j] = 2S[j]\left(k - \frac{2}{3}\right) + \left(k - \frac{2}{3}\right)^2 4^{-(j+1)}. \tag{17b}$$

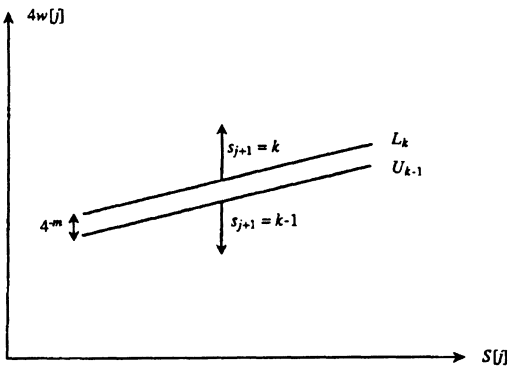


Fig. 2. Continuity.

B. Continuity Condition and Overlap Between Selection Intervals

A second requirement for the selection interval is the *continuity condition*. It states that for any value of $4w[j]$ between $4\underline{B}[j]$ and $4\bar{B}[j]$ it must be possible to select *some* value for the result digit. This can be expressed as

$$U_{k-1} \ge L_k - 4^{-m}$$

where, as shown in Fig. 2, the term 4^{-m} appears because of the granularity of the representation of the residual with m radix-4 digits.

Moreover, to use estimates of $4w[j]$ and $S[j]$ for the result-digit selection, it is necessary to have an overlap between adjacent selection intervals. For the square root operation with digit-set $\{-2, -1, 0, 1, 2\}$ from (17) the overlap is

$$U_{k-1} - L_k = \frac{1}{3}(2S[j] + (2k-1)4^{-(j+1)}). \tag{18}$$

Note that the bounds, selection intervals, and the overlap depend on j.

C. Staircase Result-Digit Selection for Residual in Redundant Form

As indicated before, to have a fast recurrence step, a redundant adder is used. We now determine a staircase result-digit selection using an estimate of the partial result and an estimate of the shifted residual obtained by truncating the redundant form.

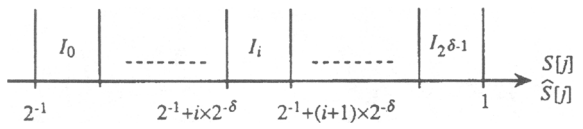


Fig. 3. Generic intervals.

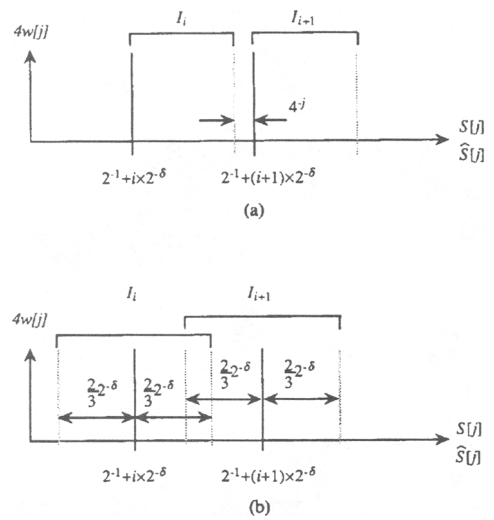


Fig. 4. Selection intervals. (a) Truncation of conventional form. (b) Truncation of signed-digit form.

As illustrated in Fig. 3, the values of the estimate of $S[j]$, called $\hat{S}[j]$, divide the range of $S[j]$ into intervals I_i (one interval per value of the estimate) so that

$$\text{If } \hat{S}[j] = 2^{-1} + i \times 2^{-\delta} \text{ then } S[j] \in I_i \qquad \text{for } 0 \le i \le 2^{\delta-1} \tag{19}$$

where δ is the number of fractional bits of the estimate $\hat{S}[j]$. Note that the value of $\hat{S}[j]$ for $i = 0$ is 2^{-1}, since the result is normalized, and that $\hat{S}[j] = 1$ for $i = 2^{\delta-1}$.

Fixed and variable estimate of S[j]: Since the result is produced one digit per iteration in signed-digit form, the following alternatives can be used to form the estimate of $S[j]$ [1] (this is in contrast to division, where the estimate is always obtained by truncating the conventional representation of the divisor):

a) Use the truncated conventional representation of $S[j]$. As discussed in Section IV, this conventional representation has to be formed on-the-fly anyhow since we use a carry-save adder for the recurrence. In this case, if $\hat{S}[j]$ is obtained by truncating $S[j]$ to δ fractional bits, then as shown in Fig. 4(a), I_i (the ith interval) is defined by

$$2^{-1} + i \times 2^{-\delta} \le S[j] < 2^{-1} + (i+1) \times 2^{-\delta}$$
$$0 \le i \le 2^{\delta-1}.$$

However, since $S[j]$ has only j fractional radix-4 digits, the upper bound of the interval is restricted so that I_i becomes

$$I_i = [2^{-1} + i \times 2^{-\delta}, 2^{-1} + (i+1) \times 2^{-\delta} - 4^{-j}). \tag{20}$$

b) Use directly the truncated signed-digit representation or

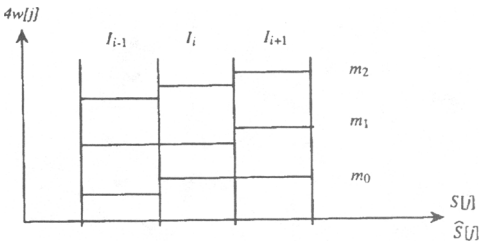


Fig. 5. Staircase result-digit selection.

(equivalently) the fixed value of $S[\lceil \delta/2 \rceil]$ (that is, the value of $S[j]$ immediately after at least δ fractional bits are produced). In this case [Fig. 4(b)], the ith interval is defined by

$$I_i = \left(2^{-1} + i \times 2^{-\delta} - \frac{2}{3} \times 2^{-\delta}, 2^{-1} + i \times 2^{-\delta} + \frac{2}{3} \times 2^{-\delta}\right). \tag{21}$$

The result-digit selection depends (slightly) on which of the two methods is used. However, since the difference is not significant, we will use method a) in our implementation.

Staircase Result-Digit Selection: Using the estimate of the result $\hat{S}[j]$, the result-digit selection is described by the set of *selection constants*

$$\{m_k(i) | 2^{-1} + i \times 2^{-\delta} \in \text{set of values of } \hat{S}[j],$$
$$k \in \{-2, -1, 0, 1, 2\}\}. \tag{22}$$

That is, there is one selection constant per value of $\hat{S}[j]$ and per value of the result digit. In terms of these selection constants, the *staircase result-digit selection* is defined by

$$s_{j-1} = k \text{ if } \hat{S}[j] = 2^{-1} + i \times 2^{-\delta} \text{ and}$$
$$m_k(i) \le \hat{w}[j] < m_{k+1}(i) \tag{23}$$

where $\hat{w}[j]$ is an estimate of the shifted residual $4w[j]$ obtained by truncating the redundant form to t fractional bits. This is illustrated in Fig. 5.

For the case of the residual represented in carry-save form, the error introduced by using the estimate (with respect to the truncated residual in conventional two's complement representation) is

$$0 \le \text{error} < 2^{-t}$$

as shown in Fig. 6(a). Consequently, the result-digit selection has to satisfy the relations [see Fig. 6(b)] [4]

$$m_k(i) \ge \max(L_k(I_i))$$
$$m_k(i) + 2^{-t} \le \min(U_{k-1}(I_i)). \tag{24}$$

Note that the relations depend on j, the iteration number; consequently, the selection constants can, in general, be different for different j.

From these expressions, the minimum overlap required for a feasible result-digit selection is

$$\min(U_{k-1}(I_i)) - \max(L_k(I_i)) \ge 2^{-t}. \tag{25}$$

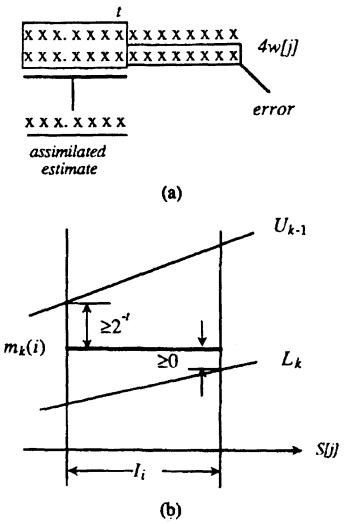


Fig. 6. (a) Error in the estimate—carry-save form. (b) Conditions on selection constants.

For method a) of formation of the estimate $\hat{S}[j]$, from (17) and (20) we get for $\min(U_{k-1}(I_i))$ and $\max(L_k(I_i))$

For $k > 0$

$$\min(U_{k-1}(I_i)) = 2(2^{-1} + i \times 2^{-\delta})\left(k - \frac{1}{3}\right) + \left(k - \frac{1}{3}\right)^2 4^{-(j+1)}$$

$$\max(L_k(I_i)) = 2(2^{-1} + (i+1) \times 2^{-\delta})\left(k - \frac{2}{3}\right) - \left(k - \frac{2}{3}\right)\left(\frac{26}{3} - k\right)4^{-(j+1)}. \quad (26a)$$

For $k \leq 0$

$$\min(U_{k-1}(I_i)) = 2(2^{-1} + (i+1) \times 2^{-\delta})\left(k - \frac{1}{3}\right) + \left(\frac{1}{3} - k\right)\left(\frac{25}{3} - k\right)4^{-(j+1)}$$

$$\max(L_k(I_i)) = 2(2^{-1} + i \times 2^{-\delta})\left(k - \frac{2}{3}\right) + \left(k - \frac{2}{3}\right)^2 4^{-(j+1)}. \quad (26b)$$

These expressions are used to determine the result-digit selection. However, since they depend on j, a different selection function might result for different j. If we want to have a single selection function, we need to develop expressions that are independent of j. For $\min(U_{k-1}(I_i))$, the term depending on j is always positive and approaching zero for large j; therefore, this term can be neglected. On the other hand, for $\max(L_k(I_i))$, the term depending on j is negative for $k > 0$ and can be neglected, but is positive for $k \leq 0$ so it cannot be neglected and we have to use its maximum value (which occurs for $j = 0$). Consequently, the corresponding expressions independent of j are as follows:

For $k > 0$

$$\min(U_{k-1}(I_i)) = 2(2^{-1} + i \times 2^{-\delta})\left(k - \frac{1}{3}\right)$$

$$\max(L_k(I_i)) = 2(2^{-1} + (i+1) \times 2^{-\delta})\left(k - \frac{2}{3}\right). \quad (27a)$$

For $k \leq 0$

$$\min(U_{k-1}(I_i)) = 2(2^{-1} + (i+1) \times 2^{-\delta})\left(k - \frac{1}{3}\right)$$

$$\max(L_k(I_i)) = 2(2^{-1} + i \times 2^{-\delta})\left(k - \frac{2}{3}\right) + \left(k - \frac{2}{3}\right)^2 4^{-1}. \quad (27b)$$

To determine whether a single selection function is possible we apply (25) to (27). The worst case is $i = 0$ and $k = -1$, resulting in

$$\min(U_{-2}(I_0)) - \max(L_{-1}(I_0)) = -\frac{8}{3}(2^{-1} + 2^{-\delta}) + \frac{10}{3}(2^{-1}) - \frac{25}{9}(4^{-1}) = -\frac{13}{36} - \frac{8}{3}2^{-\delta} \geq 2^{-t}. \quad (28)$$

Since there is no pair of values of t and δ that satisfy this inequality, no single selection function exists for all j. A possible alternative is to find a value J so that a single selection can be used for $j \geq J$ and then consider separately the cases for $j < J$.

For the case $j \geq J$, the same considerations given before produce

For $k > 0$

$$\min(U_{k-1}(I_i)) = 2(2^{-1} + i \times 2^{-\delta})\left(k - \frac{1}{3}\right)$$

$$\max(L_k(I_i)) = 2(2^{-1} + (i+1) \times 2^{-\delta})\left(k - \frac{2}{3}\right). \quad (29a)$$

For $k \leq 0$

$$\min(U_{k-1}(I_i)) = 2(2^{-1} + (i+1) \times 2^{-\delta})\left(k - \frac{1}{3}\right)$$

$$\max(L_k(I_i)) = 2(2^{-1} + i \times 2^{-\delta})\left(k - \frac{2}{3}\right) + \left(k - \frac{2}{3}\right)^2 4^{-(J+1)}. \quad (29b)$$

Introducing these expressions in (25) (for the worst case $i = 0$, $k = -1$) we get [similar to (28)]

$$\frac{1}{3} - \frac{25}{36}(4^{-J}) - \frac{8}{3}(2^{-\delta}) \geq 2^{-t}. \quad (30)$$

TABLE I
SELECTION INTERVALS FOR $j \geq 3$ ($t = 4$)

i $\hat{S}[j]$	0 8/16	1 9/16	2 10/16	3 11/16	4 12/16	5 13/16	6 14/16	7 15/16, 16/16
$ML_2(i), m\hat{U}_1(i)$	3/2, 77/48	5/3, 29/16	11/6, 97/48	2, 107/48	13/6, 39/16	7/3, 127/48	15/6, 137/48	8/3, 49/16
$ML_1(i), m\hat{U}_0(i)$	3/8, 29/48	5/12, 11/16	11/24, 37/48	1/2, 41/48	13/24, 15/16	7/12, 49/48	5/8, 53/48	2/3, 19/16
$ML^*_0(i), m\hat{U}_{-1}(i)$	-2/3, -7/16	-3/4, -23/48	-5/6, -25/48	-11/12, -9/16	-1, -29/48	-13/12, -31/48	-7/6, -11/16	-5/4, -35/48
$ML^*_{-1}(i), m\hat{U}_{-2}(i)$	-5/3, -25/16	-15/8, -83/48	-25/12, -91/48	-55/24, -33/16	-15/6, -107/48	-65/24, -115/48	-35/12, -41/16	-75/24, -133/48

A solution is $\delta = 4, J = 3, t = 3$. However, when applying the conditions (24), one of the selection constants is $-29/16$ [4]; consequently, it is necessary to use $t = 4$. Table I shows the corresponding limits of the intervals. The following notation is used

$$m\hat{U}_{k-1}(i) = \min(U_{k-1}(I_i)) - \frac{1}{16}$$

$$ML_k(i) = \max(L_k(I_i)).$$

As shown in the expressions (29), for $k \leq 0$ the expressions for $ML_k(i)$ contain the term $(k - 2/3)^2 4^{-4}$. This term has the following values:

k	0	-1
$\frac{(k-2/3)^2}{256}$	1/576	1/90

Since these values are small (compared to $2^{-t} = 1/16$), it is simpler to present in the table

$$ML^*_k(i) = \max(L_k(I_i)) - \left(k - \frac{2}{3}\right)^2 4^{-4}$$

instead of $ML_k(i)$, with the restriction that the selection constant $m_k(i)$ cannot be equal to $ML^*_k(i)$.

Now we have to consider the cases $j < 3$. One possible approach is to have an initial PLA to determine from a truncated x the value of S [3]. Another possibility is to analyze the cases $j < 3$ and then find a combined result-digit selection (which might depend on the value of j). As indicated in the Introduction, unlike other reported implementations, we follow this second approach because it potentially results in a simpler implementation.

For all cases, we use $\delta = 4$ and $t = 4$ to match with the case $j \geq 3$.

As indicated in expression (1), since the maximum value of the radix-4 digit is 2, it is necessary to have

$$s_0 = 1$$

to be able to represent values of $s \geq 2/3$. Consequently, $S[0] = 1$. Moreover, the values of s_1 are restricted to the set $s_1 \in \{0, -1, -2\}$ (to have $s < 1$).

Therefore, for $j = 0$ and $S[0] = 1$ we obtain

$$\hat{U}_{-1} = -2 \times 1 \times (1/3) + (1/9)(1/4) - 1/16 = -(101/144)$$

$$L_0 = -2 \times 1 \times (2/3) + (4/9)(1/4) = -(44/36)$$

TABLE II
SELECTION INTERVALS FOR $j = 1$

i $\hat{S}[1] = S[1]$	0 1/2	4 3/4	8 1
$L_2(i), \hat{U}_1(i)(\times 144)$	208, 256	304, 376	400, 496
$L_1(i), \hat{U}_0(i)(\times 144)$	49, 91	73, 139	97, 187
$L_0(i), \hat{U}_{-1}(i)(\times 144)$	-	-140, -80	-188, -104
$L_{-1}(i), \hat{U}_{-2}(i)(\times 144)$	-	-335, -281	-455, -377

$$\hat{U}_{-2} = -2 \times 1 \times (4/3) + (16/9)(1/4) - 1/16$$

$$= -(329)/(144)$$

$$L_{-1} = -2 \times 1 \times (5/3) + (25/9)(1/4) = -(95)/(36).$$

For $j = 1$ we can have $s_2 \in \{-2, -1, 0, 1, 2\}$. Moreover, since $s_1 \in \{-2, -1, 0\}$ and $S[0] = 1$ the only possible values of $S[1]$ are 1/2, 3/4, and 1. Since $\delta = 4$ (because of the case $j \geq 3$), the estimate $S[1]$ coincides with $S[1]$ and we need consider only the values of U_{k-1} and L_k for 1/2, 3/4, and 1 and not consider min or max values in intervals. The corresponding values of L_k and U_{k-1} are given in Table II. Note that it is not possible to select $s_2 < 0$ when $S[1] = 1/2$, because this would make $S[2] < 1/2$.

For $j = 2$, since we use $\delta = 4$ and the granularity of $S[2]$ is 1/16 again $\hat{S}[2] = S[2]$, so we use exact values of $S[2]$ instead of intervals for the computation of L_k and U_{k-1}. These values are shown in Table III.

We now need to obtain a single result-digit selection for $j = 0$, $j = 1$, $j = 2$, and $j \geq 3$. To do this we combine all previous tables into Table IV.

From Table IV we can see that for all entries except for m_{-1} (8) it is possible to have a single selection function for all values of j; that is, for each pair (k, i) it is possible to find a selection constant that is inside the allowed intervals for all j. The single nonconforming case is $k = -1$, $i = 8$, and $j = 0$, for which the selection interval is $[-1520/576, -1216/576]$ which does not overlap with the corresponding intervals for $j \neq 0$. We have chosen to solve this problem by changing the estimate for $j = 0$ ($\hat{S}[0]$) from its normal value 1 to either 12/16 or 13/16; this satisfies the only two cases which are significant for $j = 0$, namely $m_0(8)$ and $m_{-1}(8)$.

In addition, to reduce the number of bits required for the estimate $\hat{S}[j]$, we convert $\hat{S}[j] = 1$ into $\hat{S}[j] = 15/16$ (for $j \neq 0$), that is we fold interval $i = 8$ onto $i = 7$; this can

TABLE III
SELECTION INTERVALS FOR $j = 2$

i	0	1	2	3	4	5	6	7	8
$\hat{S}[2] = S[2]$	8/16	9/16	10/16	11/16	12/16	13/16	14/16	15/16	16/16
$L_2, U_1(\times 576)$	784, 922	880, 1042	976, 1152	1072, 1262	1168, 1372	1264, 1482	1360, 1592	1456, 1702	1552, 1812
$L_1, U_0(\times 576)$	193, 325	217, 373	241, 421	265, 469	289, 517	333, 565	357, 613	381, 661	405, 709
$L_0, \hat{U}_{-1}(\times 576)$	-380, -254	-428, -278	-476, -302	-524, -326	-572, -350	-620, -374	-668, -398	-716, -422	-754, -444
$L_{-1}, \hat{U}_{-2}(\times 576)$	-935, -815	-1045, -911	-1155, -1007	-1265, -1103	-1375, -1199	-1485, -1295	-1595, -1391	-1696, -1487	-1815, -1583

TABLE IV
SELECTION INTERVALS AND SELECTION CONSTANTS FOR ALL VALUES OF j

i	0	1	2	3	4	5	6	7	8
$\hat{S}[j]$	8/16	9/16	10/16	11/16	12/16	13/16	14/16	15/16	16/16
$ML_2(i), m\hat{U}_1(i)$ (×576)									
$j \geq 3$	864, 924	960, 1044	1056, 1164	1152, 1284	1248, 1404	1334, 1524	1440, 1644	1536, 1764	1536, 1764
$j = 2$	784, 922	880, 1042	976, 1152	1072, 1262	1168, 1372	1264, 1482	1360, 1592	1456, 1702	1552, 1812
$j = 1$	832, 1024				1216, 1504				1600, 1984
$j = 0$									-
$m_2(i)$	3/2	7/4	2	2	9/4	5/2	5/2	11/4	3
$m_2(i) \times 576$	864	1008	1152	1152	1296	1440	1440	1584	1728
$ML_1(i), m\hat{U}_0(i)$ (×576)									
$j \geq 3$	216, 348	240, 396	264, 444	288, 492	312, 540	336, 588	360, 636	384, 684	384, 684
$j = 2$	193, 325	217, 373	241, 421	265, 469	289,517	333, 565	357, 613	381, 661	405, 709
$j = 1$	196, 364				292, 556				388, 748
$j = 0$									-
$m_1(i)$	1/2	1/2	1/2	1/2	3/4	3/4	1	1	1
$m_1(i) \times 576$	288	288	288	288	432	432	576	576	576
$ML^*_0(i), m\hat{U}_{-1}(i)$ (×576)									
$j \geq 3$	-384, -252	-432, -306	-570, -300	-528, -324	-576, -348	-624, -372	-672, -396	-720, -420	-720,-420
$j = 2$	-, -	-428, -278	-476, -302	-524, -326	-572, -350	-620, -374	-668, -398	-716, -422	-754, -444
$j = 1$	-				-560, -320				-752, -416
$j = 0$									-704, -404
$m_0(i)$	-1/2	-5/8	-3/4	-3/4	-3/4	-1	-1	-1	-1
$m_0(i) \times 576$	-288	-360	-432	-432	-432	-576	-576	-576	-576
$ML^*_{-1}(i), m\hat{U}_{-2}(i)$ (×576)									
$j \geq 3$	-960, -900	-1080, -996	-1200,-1092	-1320,-1188	-1440,-1284	-1560,-1380	-1680,-1476	-1800,-1596	-1800,-1596
$j = 2$	-,-	-1045,-911	-1155,-1007	-1265,-1103	-1375,-1199	-1485,-1295	-1595,-1391	-1696,-1487	-1815,-1583
$j = 1$	-				-1340,-1124				-1820,-1408
$j = 0$									-1520,-1216
$m_{-1}(i)$	-13/8	-7/4	-2	-17/8	-9/4	-5/2	-11/4	-23/8	*
$m_{-1}(i) \times 576$	-936	-1008	-1152	-1224	-1296	-1440	-1584	-1656	

be done because the selection intervals are satisfied. Therefore, the estimate used in the selection $\hat{S} = (\hat{S}_1, \hat{S}_2, \hat{S}_3, \hat{S}_4)$ is obtained as follows:

$$(\hat{S}_1, \hat{S}_2, \hat{S}_3, \hat{S}_4) = \begin{cases} (1, 1, 0, -) & \text{if } (j = 0) \\ (1, 1, 1, 1) & \text{if } (A_0 = 1) \text{ and } (j \neq 0) \\ (1, A_2, A_3, A_4) & \text{if } (j \neq 0) \end{cases}$$

where $(A_0, A_1, A_2, A_3, A_4)$ are the most significant bits of A, the conventional representation of $S[j]$ (as discussed in Section IV), and for $j = 0$, $A_0 = 1$ and $A_2 = A_3 = A_4 = 0$.

The resulting selection function is given in Table V and its implementation shown in Fig. 7. From the possible selection constants, those which minimize the number of fractional bits in their representation are chosen. Since the selection constants are of the form $D \times 2^{-3}$ (D integer), 3 fractional bits of the estimate of the shifted residual are used in the result-digit

TABLE V
RESULT-DIGIT SELECTION FOR ALL j

i	0	1	2	3	4	5	6	7
$\hat{S}[j]$	8/16	9/16	10/16	11/16	12/16	13/16	14/16	15/16
$m_2(i)$	3/2	7/4	2	2	9/4	5/2	5/2	11/4
$m_1(i)$	1/2	1/2	1/2	1/2	3/4	3/4	1	1
$m_0(i)$	-1/2	-5/8	-3/4	-3/4	-3/4	-1	-1	-1
$m_{-1}(i)$	-13/8	-7/4	-2	-17/8	-9/4	-5/2	-11/4	-23/8

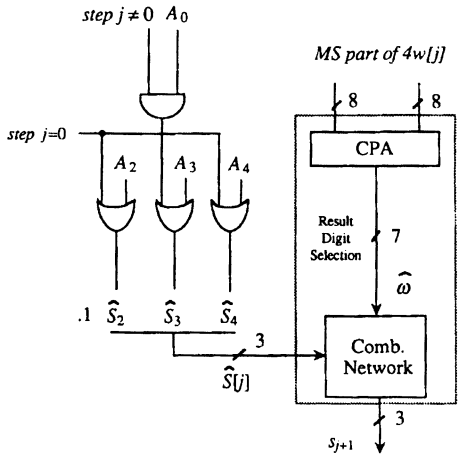


Fig. 7. Result-digit selection implementation.

selection. Moreover, since $t = 4$, 4 fractional bits of the shifted carry-save residual are used, resulting in a total of 8 bits (4 integer bits and 4 fractional bits).

Alternative implementations exist for the combinational network of the result-digit selection. One possibility is to use a ten-input/three-output PLA; on the other hand, some preprocessing can be done to reduce the size of the PLA, as proposed for example in [5].

IV. Generation of Adder Input F

As part of the implementation of the recurrence (9) it is necessary to form the adder input F with value

$$F[j] = -2S[j]s_{j+1} - s_{j+1}^2 4^{-(j+1)}.$$

Since the digit of the result is produced in signed-digit form, the partial result $S[j]$ is also in this form. However, for the case we are considering, which uses a carry-save adder, the input F has to be in two's complement representation. Consequently, $S[j]$ is converted to this form on-the-fly using a variation of the scheme presented in [2]. It requires that two conditional forms $A[j]$ and $B[j]$ are kept, such that

$$A[j] = S[j]$$

$$B[j] = S[j] - 4^{-j}.$$

These forms are updated with each result-digit as follows:

$$A[j+1] = \begin{cases} A[j] + s_{j+1}4^{-(j+1)} & \text{if } s_{j+1} \geq 0 \\ B[j] + (4 - |s_{j+1}|)4^{-(j+1)} & \text{otherwise.} \end{cases} \quad (31a)$$

$$B[j+1] = \begin{cases} A[j] + (s_{j+1} - 1)4^{-(j+1)} & \text{if } s_{j+1} > 0 \\ B[j] + (3 - |s_{j+1}|)4^{-(j+1)} & \text{otherwise.} \end{cases} \quad (31b)$$

The implementation of this conversion requires two registers for A and B, appending of one digit, and loading. For controlling this appending and loading, a shift register K is used, containing a moving 1. This implementation is shown in Fig. 8. In terms of these forms, the value of F is given by the following expressions:

For $s_{j+1} > 0$

$$\begin{aligned} F[j] &= -2S[j]s_{j+1} - s_{j+1}^2 4^{-(j+1)} \\ &= -(2A[j] + s_{j+1}4^{-(j+1)})s_{j+1}. \end{aligned} \quad (32a)$$

For $s_{j+1} < 0$

$$\begin{aligned} F[j] &= 2S[j]|s_{j+1}| - s_{j+1}^2 4^{-(j+1)} \\ &= 2(B[j] + 4^{-j})|s_{j+1}| - s_{j+1}^2 4^{-(j+1)} \\ &= (2B[j] + (4 - |s_{j+1}|)4^{-(j+1)})|s_{j+1}|. \end{aligned} \quad (32b)$$

Note that these expressions are obtained by concatenation and multiplication by a radix-4 digit. The resulting bit-strings are given in Table VI, where $a \cdots aa$ and $b \cdots bb$ are the bit-strings representing $A[j]$ and $B[j]$, respectively (shifted one position). The location of the trailing string is also controlled by the moving 1 of register K.

Fig. 8 also shows a block to perform on-the-fly rounding, as described in [3].

V. Overall Algorithm, Implementation, and Timing

The overall algorithm is as follows (not including rounding):

begin Square root
Initialization
$w[0] = x - 1$ *since $w[-1] = S[-1] = 0$ and $s_0 = 1$ (Load x and make 1 the sign position)*
$A[0] \leftarrow 1.000...000$ *$S[0] = 1$*
$B[0] \leftarrow 0.000...000$ *$B[0] = A[0] - 1$*
$K[0] \leftarrow 0.100...000$
Iterations
for $j = 0$ **to** m
begin
$s_{j+1} =$ SELECT($\hat{w}[j]$, $\hat{S}[j]$) *see Table V*
$F[j] = f(A[j], B[j], s_{j-1})$ *see Table VI*
$w[j+1] \leftarrow 4w[j] + F[j]$
$A[j+1] \leftarrow g_a(A[j], B[j], s_{j+1})$ *see expression (31a)*
$B[j+1] \leftarrow g_b(A[j], B[j], s_{j+1})$ *see expression (31b)*
$K[j+1] \leftarrow$ *shift-right* $(K[j])$
end for
Result
$s = S[m] = A[m]$
end Square root

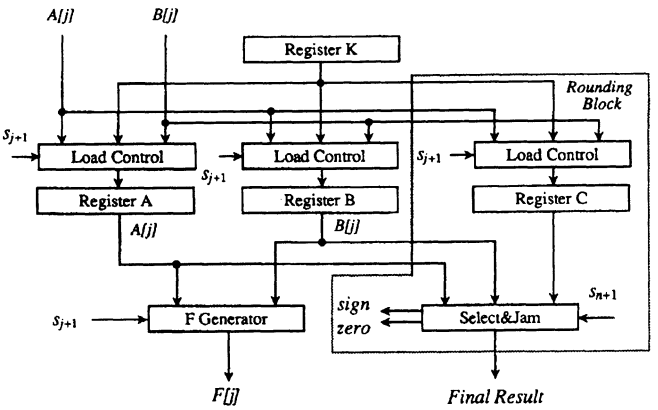


Fig. 8. Network for generating F and rounding.

TABLE VI
GENERATION OF $F[j]$

	$F[j]$		
s_{j+1}	Value	Value	Bit-string
0	0	0	$0\cdots00000$
1	$-2S[j]-4^{-(j+1)}$	$-2A[j]-4^{-(j+1)}$	$\bar{a}\cdots\bar{a}\bar{a}111$
2	$-4S[j]-4\times4^{-(j+1)}$	$-4A[j]-4\times4^{-(j+1)}$	$\bar{a}\cdots\bar{a}1100$
-1	$2S[j]-4^{-(j+1)}$	$2B[j]+7\times4^{-(j+1)}$	$b\cdots bb111$
-2	$4S[j]-4\times4^{-(j+1)}$	$4B[j]+12\times4^{-(j+1)}$	$b\cdots b1100$

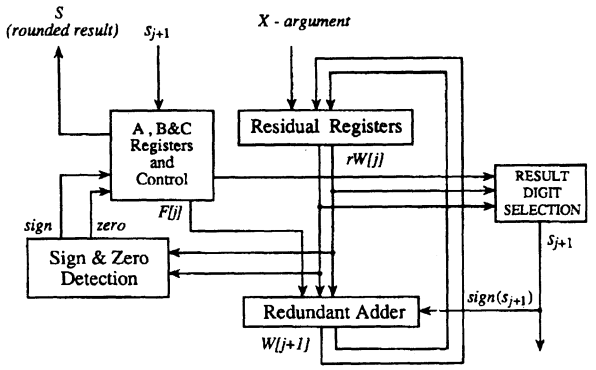


Fig. 9. Block diagram of the square root scheme (mantissa part).

The overall implementation at the block-diagram level is shown in Fig. 9. The cycle time is

$$
\begin{aligned}
T_{\text{cycle}} &= t_{\text{result\_digit\_select}} && \{\text{8-bit CPA} + \text{10-input comb. net}\}\\
&+t_{F\_\text{generate}} && \{\text{4-to-1 multiplexer}\}\\
&+t_{\text{CSA}} && \{\text{3-to-2 carry-save adder}\}\\
&+t_{\text{load}} && \{\text{register loading}\}.
\end{aligned}
$$

This is comparable to the cycle time of a radix-4 division with carry-save adder.

VI. Conclusions

We have shown a radix-4 square root algorithm that does not require an initial PLA. More precisely, this PLA is replaced by the four gates of Fig. 7. This is of theoretical interest and also produces a simpler implementation. It should be noted that the number of iterations required in this algorithm is the same as in those that use an initial PLA. That is, the fact that the initial bits of the result are obtained from the PLA does not reduce the number of iterations, since the residual has to be obtained from the iterations. Consequently, the simplification in implementation is obtained without time penalty.

Acknowledgment

We thank P. Montuschi for helpful suggestions.

References

[1] L. Ciminiera and P. Montuschi, "Higher radix square rooting," Intern. Rep., Politecnico di Torino, Dipartimento di Automatica e Informatica, I.R. DAI/ARC 4-87, Dec. 1987.

[2] M. D. Ercegovac and T. Lang, "On-the-fly conversion of redundant into conventional representations," *IEEE Trans. Comput.*, vol. C-36, no. 7, pp. 895-897, July 1987.

[3] ——, "On-the-fly rounding for division and square root," in *Proc. 9th Symp. Comput. Arithmetic*, 1989, pp. 169-173.

[4] ——, "Division and square root algorithms and implementations," monograph in preparation.

[5] J. Fandrianto, "Algorithm for high speed shared radix-4 division and radix-4 square root," in *Proc. 8th Symp. Comput. Arithmetic*, 1987, pp. 73-79.

[6] J. B. Gosling and C. M. S. Blakeley, "Arithmetic unit with integral division and square-root," *IEE Proc.*, vol. 134, pt. E, no. 1, pp. 17-23, Jan. 1987.

[7] S. Kuninobu *et al.*, "Design of high speed MOS multiplier and divider using redundant binary representation," in *Proc. 8th Int. Symp. Comput. Arithmetic*, 1987, pp. 80-86.

[8] P. Montuschi and L. Ciminiera, "On the efficient implementation of higher radix square root algorithms," in *Proc. 9th Symp. Comput. Arithmetic*, Sept. 1989, pp. 154-161.

[9] M. B. Vineberg, "A radix-4 square-rooting algorithm," Rep. 182, Dep. Comput. Sci., Univ. of Illinois, Urbana-Champaign, June 1965.

[10] J. H. Zurawski and J. B. Gosling, "Design of a high-speed square root multiply and divide unit," *IEEE Trans. Comput.*, vol. C-36, pp. 13-23, Jan. 1987.

167 MHz Radix-4 Floating Point Multiplier

Robert K. Yu and Gregory B. Zyner

SPARC Technology Business, Sun Microsystems Inc.
Sunnyvale, California

Abstract - **An IEEE floating point multiplier with partial support for subnormal operands and results is presented. Radix-4 or modified Booth encoding and a binary tree of 4:2 compressors are used to generate the 53x53 double-precision product. Delay matching techniques were used in the binary tree stage and in the final addition stage to reduce cycle time. New techniques in rounding and sticky-bit generation were also used to reduce area and timing. The overall multiplier has a latency of 3 cycles, a throughput of 1 cycle, and a cycle time of 6.0ns. This multiplier has been implemented in a 0.5um static CMOS technology in the *UltraSPARC* RISC microprocessor.**

I. Introduction

Multiplier units are commonly found in digital signal processors and, more recently, in RISC-based processors[1]. Double-precision floating point operations involve the inherently slow operation of summing 53 partial products together to produce the product. IEEE-compliant multiplication also involves the correct rounding of the product, adjustment to the exponent, and generation of correct exception flags. Multiplier units embedded in modern RISC-based processors must also be pipelined, small, and *fast*. Judicious functional and physical partitioning are needed to meet all these requirements[5][6][14]. In this paper, Section II describes the partial subnormal support. Section III describes details of the implementation, including the 4:2 compressor design, the binary tree composition, final addition, rounding, and sticky-bit generation. Finally, Section IV concludes this paper.

II. Subnormal Operations

Multiplier units that handle subnormal or denormal operands and results often require the determination of leading zeros, adjustments to mantissas (shifting) and exponent, and rounding. Overall timing and area are affected when subnormal operations are fully supported. However, by introducing a modest amount of hardware to compare only the value of the exponents, we can provide support for a large subset of the subnormal operations. This partial subnormal support does not require detection of leading zeros, adjustments of subnormal mantissas, and does not introduce an extra cycle penalty.

Figure 1 shows the floating point data formats, and Figure 2 shows their value definitions according to IEEE standards. The value of the mantissa is formed by concatenating the implicit bit with the fraction. For normalized values, the implicit bit is one, and for subnormal values, the implicit bit is zero. Note that for subnormal values, the convention is to have the exponent field zeroed, but the *value of the exponent* is taken to be one.

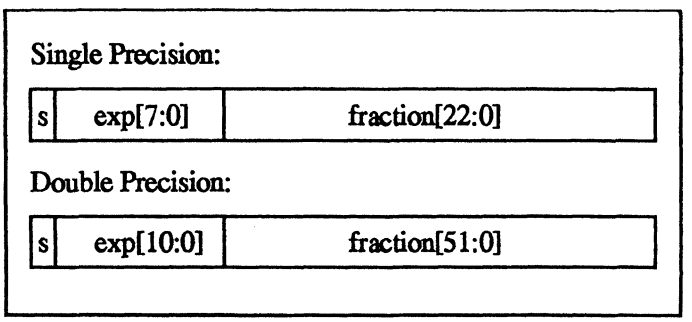


Fig. 1. Floating point formats.

s = sign
e = biased exponent
f = fraction
E = number of bits in exponent (8 for single, 11 for double)
F = number of bits in fraction (23 for single, 52 for double)
B = exponent bias (127 for single, 1023 for double)

Normalized Value ($0 < e < 2^E-1$):
$$(-1)^s \times 2^{e-B} \times 1.f$$

Subnormal Value ($e = 0$):
$$(-1)^s \times 2^{1-B} \times 0.f$$

Zero:
$$(-1)^s \times 0$$

Fig. 2. Floating point format definition.

In multiplication, the resultant exponent e_r is calculated by:

$$e_r = e_1 + e_2 - B - z_1 - z_2 \quad (1)$$

where e_1 and e_2 are the biased exponents, and z_1 and z_2 are the leading zeros in the mantissas to the multiplicand and mul-

Reprinted from *Proceedings of the 12th Symposium on Computer Arithmetic*, pp. 149-154, July 1995.

tiplier, respectively. We can simplify Equation 1 by noting that if both operands are subnormal, then the value of e_r underflows to a value no greater than $-B$, and cannot be represented in the given precision[1]. We can therefore impose the constraint that only one operand can be subnormal without the loss of generality and rewrite this equation as:

$$e_r = e_1 + e_2 - B - z \tag{2}$$

where z is the number of leading zeros of the subnormal operand, if any. We further note that if the resulting value e_r is less than one, then the amount by which the resulting mantissa needs to be right-shifted to set $e_r = 1$ is:

$$rshift = 1 - e_r \tag{3}$$

If the *rshift* value is greater than the number of bits in the mantissa, then again we underflow beyond the dynamic range of the given precision. We define this condition as "extreme underflow."[2] Specifically, if

$$rshift \geq (F + 3) \tag{4}$$

then the mantissa will be right-shifted to the sticky-bit position or beyond and the resulting mantissa is either zero or the smallest subnormal number, depending on the rounding mode. The two cases where rounding will produce the smallest subnormal number are 1) rounding to plus infinity and the result is positive and 2) rounding to minus infinity and the result is negative. All other rounding modes including rounding to nearest and rounding to zero produces zero as the result.

Rewriting Equation 4 in terms of e_r, extreme underflow occurs when

$$e_r \leq -(F + 2) \tag{5}$$

We note from Equation 2 that e_r requires the detection of leading zeros z. If we ignore z altogether, which greatly simplifies the implementation, then Equation 5 becomes a conservative criterion for extreme underflow. That is, using only

$$e_r = e_1 + e_2 - B \tag{6}$$

to determine extreme underflow ignores those cases where extreme underflow would occur because of leading zeros present in the subnormal mantissa.

Equation 5 and Equation 6 form the basis used to provide partial subnormal support in this design. If Equation 5 is satisfied with the appropriate rounding mode, then the multiplier generates a zero result. The multiplier does not support the case where Equation 5 is not satisfied and a subnormal operand is encountered.

1. Does not apply to single precision multiplication resulting in a double precision result, or "fsmuld".
2. Sometimes loosely referred to as "gross underflow".

III. Implementation

A. Folded 3-Stage Pipeline

The multiplier operates over three stages. In the first stage, the multiplicand and multiplier operands go through the radix-4 encoding and multiply tree[3][4][7]. The intermediate summations of partial products and the result of the tree are in carry-save format. In the second stage, a conditional-sum adder is used to determine the product, converting the result from carry-save to binary form. In the third stage, rounding and flag generation is performed. The multiplier has a single-cycle throughput and a three-cycle latency.

B. Stage 1: Multiply Tree

1) Interleaved Binary Tree with Delay Matching

The first stage of a 53x53 bit multiplication of the mantissas is performed by a radix-4 Booth encoded binary tree of 4:2 compressors, or 5:3 counters. Figure 3 shows a schematic of the binary tree. The encoding scheme produces 27 partial products which are generated at blocks 0,1,3,4,7,8, and 10. Since 4:2 compressors are used, each block generates 4 partial products, except for block 10 which generates 3.

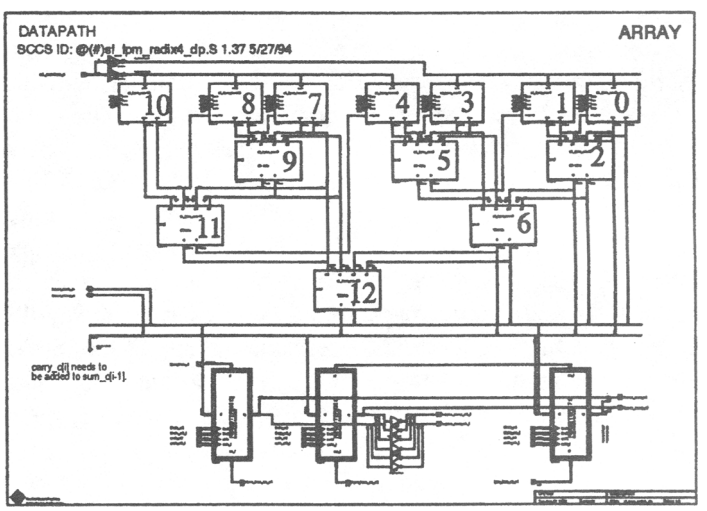


Fig. 3. Schematic of radix-4 binary tree.

Unlike traditional implementations, where inputs flow starting from the top and side of the tree to the bottom of the tree, this implementation has the multiplier and staged results placed on the same side of the tree. That is, pipeline register are embedded in the tree and are routed to the same side as the multiplicand, as shown in Figure 4. The advantage of this approach is to reduce interconnect lengths and to push some of the interconnect delay to the next stage.

The complexity of a binary tree does not lend itself to a straight-forward layout[2]. To minimize the delay through the tree due to interconnects, the placement of the rows of partial product generators and adders are done such that wire lengths are balanced among the rows. Both the vertical distance and

the horizontal distance, which comes about because the tree is "left-justified" and is significant in some cases, were taken into account.

Table I shows the vertical row distances and horizontal bit distances between cells on different rows. The critical path through the array involve the rows with large horizontal shifts, namely rows 0, 2, 6, and 12. These rows have been placed close together to reduce this path. Figure 4 shows the placement used[8].

TABLE I
DISTANCE BETWEEN CELLS

Row Transition	*Horizontal Distance*	*Vertical Distance*
0 -> 2	9	1
1 -> 2	1	2
3 -> 5	9	1
4 -> 5	1	3
7 -> 9	9	1
8 -> 9	1	2
2 -> 6	17	3
5 -> 6	1	3
9 -> 11	3	3
10 -> 11	0	7
6 -> 12	18	4
11 -> 12	0	4

2) Folded Adder Rows

Another problem presented by the irregular tree structure is the differing number of bits in each row, which varied from 61 to 76 bits. In order to reduce the area of the tree, some of the adders in the larger rows were "folded" to rows with fewer cells. The folding was done such that timing was not affected. More folding could be done but not without impacting the critical path through the tree.

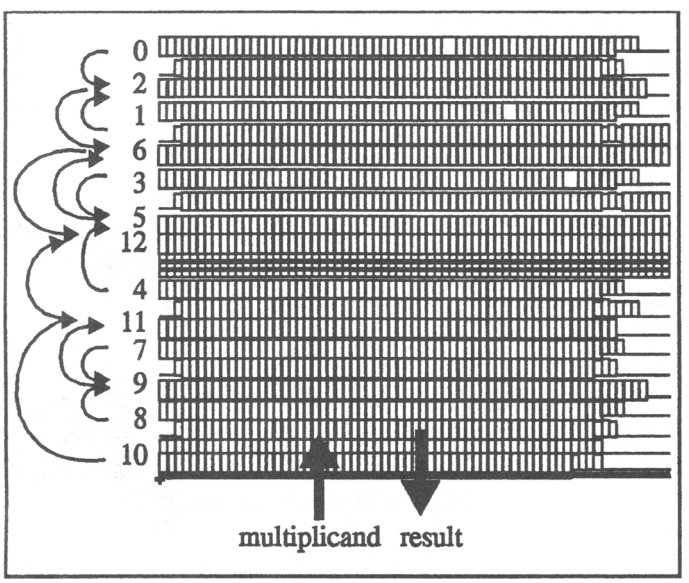


Fig. 4. Block diagram of binary tree showing the ordering of the rows used to balance the interconnect delays.

3) 4:2 Compressor Design

The 4:2 compressor schematic is shown in Figure 5. This adder takes 5 inputs {*x3, x2, x1, x0,* and *cin*} and generates 3 outputs {*carry, cout,* and *sum*}. All inputs and the *sum* output have a weight of one, and two outputs *carry* and *cout* have a weight of two[10][12]. That is,

$$2^0 \cdot (x3 + x2 + x1 + x0 + cin) = 2^1 \cdot (carry + cout) + 2^0 \cdot sum \quad (7)$$

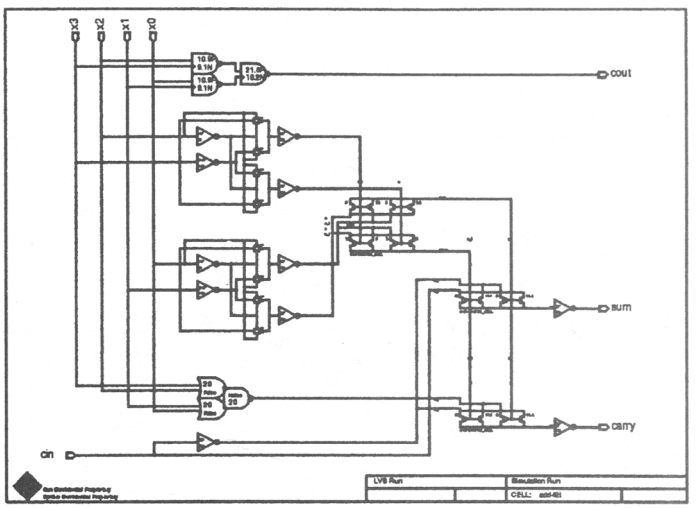


Fig. 5. Circuit schematic of 4:2 compressor.

The 4:2 compressor has been designed according to Table II. Since the *cout* signal is independent of the *cin* input and only dependent on the *x* inputs, a row of such adders hooked up together as shown in Figure 6 will not exhibit any

TABLE II
TRUTH TABLE FOR 4:2 COMPRESSOR

x3	*x2*	*x1*	*x0*	*cout*	*carry*	*sum*
0	0	0	0	0	0	*cin*
0	0	0	1	0	*cin*	$\overline{cin}$
0	0	1	0	0	*cin*	$\overline{cin}$
0	0	1	1	1	0	*cin*
0	1	0	0	0	*cin*	$\overline{cin}$
0	1	0	1	0	1	*cin*
0	1	1	0	0	1	*cin*
0	1	1	1	1	*cin*	$\overline{cin}$
1	0	0	0	0	*cin*	$\overline{cin}$
1	0	0	1	0	1	*cin*
1	0	1	0	0	1	*cin*
1	0	1	1	1	*cin*	$\overline{cin}$
1	1	0	0	1	0	*cin*
1	1	0	1	1	*cin*	$\overline{cin}$
1	1	1	0	1	*cin*	$\overline{cin}$
1	1	1	1	1	1	$\overline{cin}$

rippling of carries from *cin* to *cout*. The adder has also been designed such that the delay from *xi* to *sum* or *carry* is approximately the same as the combined delay from *xi* to *cout*, and

cin to *sum* or *carry* of an adjacent adder.

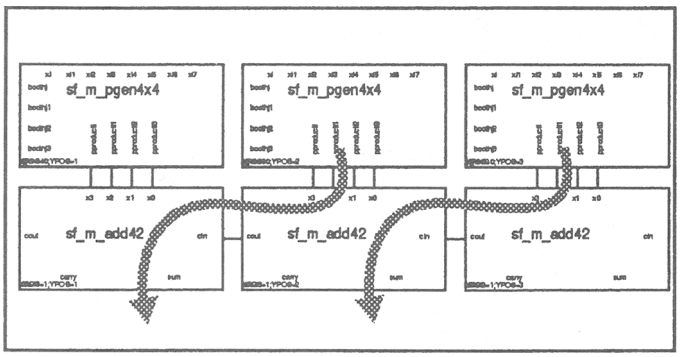


Fig. 6. Interconnect of 4:2 compressors with no horizontal ripple carry.

C. Stage 2: Final Addition

1) Optimized Conditional-Sum Adder

The final add stage of the multiplier makes use of a 52-bit conditional sum adder that is partitioned for minimum delay. Figure 7 shows a block diagram of the recursive structure of the conditional sum adder. As shown in the diagram, an N-bit conditional sum adder is made up of two smaller conditional sum adders, one that is j-bits, and one that is N-j bits wide. Two 2:1 muxes are used to output the upper sum and carry results; these outputs are selected by the carries from the lower j-bit adder. Typically the selects to the muxes are buffered up to handle the capacitive loading due to large fanouts. These smaller adders are, in turn, made up of smaller conditional sum adders[11].

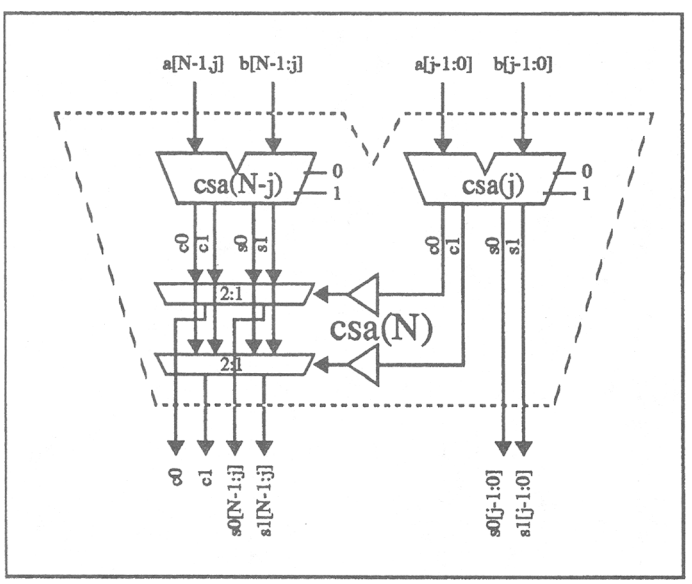


Fig. 7. Recursive structure of conditional sum adder.

The delay through this adder is affected by how the adders are partitioned. We define the delay for an N-bit adder partitioned at position *j* as *T(N,j)*:

$$T(N,j) = \max[T_{opt}(j) + T_{buf}(N-j) + T_{sel}, T_{opt}(N-j) + T_{mux}] \quad (8)$$

In Equation 8, $T_{opt}(i)$ is the optimal delay for an i-bit adder, T_{buf} is the buffer delay and is a function of the number of bits on the left adder, or N-j, T_{sel} is the select to out delay of the mux, and T_{mux} is the data to out delay of the mux. The minimum delay for an N-bit adder $T_{opt}(N)$ is simply:

$$T_{opt}(N) = min\,[T(N,j)] \quad (9)$$

where *j* varies from 1 to *N-1*. The problem of finding $T_{opt}(N)$ is a recursive min-max problem and lends itself well to an efficient dynamic programming solution developed internally for this implementation.

2) Sticky-Bit Generation from Carry-Save Format

The *sticky* bit, which is needed to perform the correct rounding, is generated in the second stage. Typically, the lower 51 bits in carry-save format are summed and OR'ed together. However, in our technique, we are able to generate the *sticky* bit directly from the outputs of the tree in carry-save format without the need for any 51-bit adder to generate the sum beforehand, resulting in significant timing and area savings. We define:

$$p_i = s_i \oplus c_i \quad (10)$$

$$h_i = s_i + c_i \quad (11)$$

$$t_i = p_i \oplus h_{i-1} \quad (12)$$

where s_i and c_i are the sum and carry outputs from the tree. The *sticky* bit is then computed directly by using a ones-detector[9]:

$$sticky = t_0 + t_1 + \ldots + t_{50} \quad (13)$$

D. Stage 3: Rounding

1) Using Conditional Sum Adders

By using a conditional sum adder to generate both the sum and sum+1, we remove the need for an incrementer to perform the rounding operation. Only multiplexing is needed to select the correct result after rounding[13].

2) Overflow After Rounding

In double precision multiplication, after the 106-bit product (in carry-save format) has been generated by the array, the decimal point occurs between bits 104 and 103, and only the upper 53-bits are used for the mantissa result. The lower 53-bits are needed only to perform the correct rounding. After rounding is performed, either bits 105-53 or bits 104-52 are used depending on the value of bit 105 or MSB. If this MSB is set, then the mantissa is taken from bits 105-53, and the value

of the exponent is incremented. Otherwise, if the MSB is not set, then bits 104-52 are used, and the exponent is not incremented. Note that the rounding itself may propagate to set the MSB; this is the case of overflow *after* rounding. Figure 8 shows how the mantissa is selected from the array result depending on bit 105 after rounding. In the figure, L, G, R, and S represents the LSB, guard, round, and sticky bits, respectively.

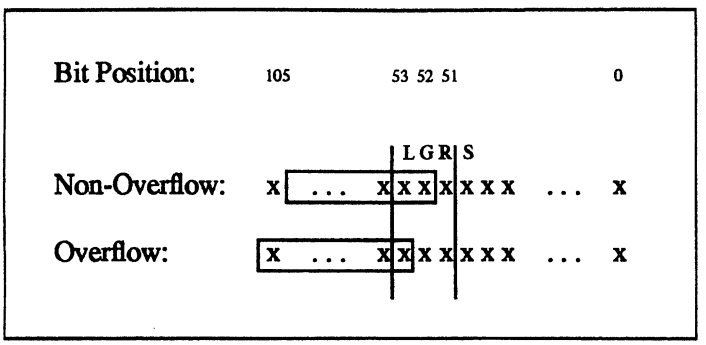


Fig. 8. Mantissa selection for overflow and non-overflow.

Figure 9 shows a block diagram of the rounding datapath and logic. The dotted line represents pipeline registers: the final add operation and rounding are done in separate stages. The lower 50 bits from the array are used to generate two signals: c51and S. The c51 signal is the carry into bit 51. Bits 53 through 51 along with c51 are added to create the L, G, and R bits, and the rest of the bits 105:54 are added using a conditional sum adder to form two results sum0[105:54] and sum1[105:54].

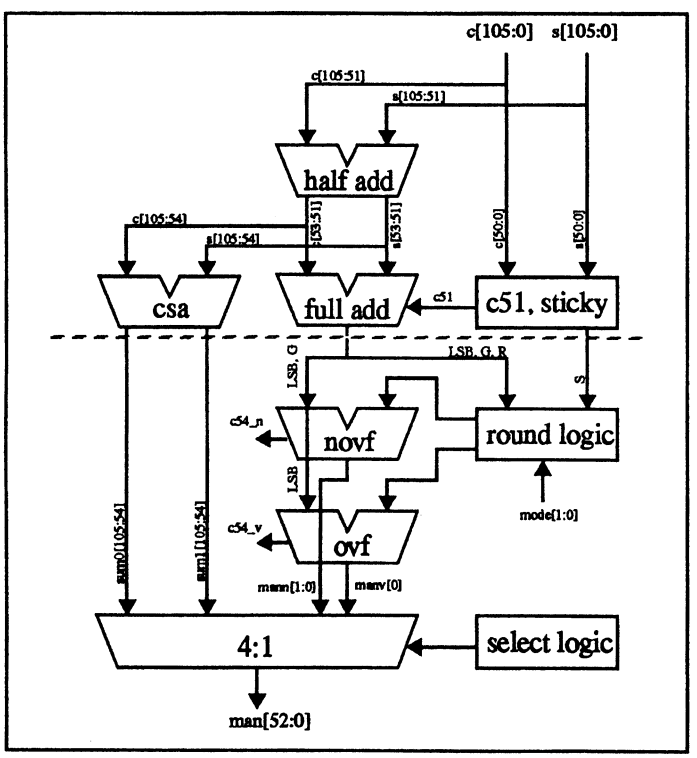


Fig. 9. Rounding.

Because of the c51 signal and a rounding that may occur at bits 53 or 52, there is a possibility of introducing two carries into bit position 54. To ensure that only one carry is propagated into bit 54, a row of half-adders is used at bits 105 through 51.

To correctly handle the case of overflow *after* rounding, our implementation makes use of two adders, ovf and novf, to generate the signals c54_v and c54_n, respectively, which are needed by the final selection logic. The c54_v and c54_n are the carries into bit position 54 assuming an overflow and non-overflow, respectively. The L, G, R, S, and rounding mode bits are used by the round logic to generate two rounding values. One value assumes a mantissa overflow, and the other assumes no mantissa overflow. These rounding bits are added to the L and {L,G} bits to form the lower one and two bits of the resulting mantissa for overflow (manv) and non-overflow (mann), respectively.

3) Final Selection

The final select logic combines the appropriate sum0 and sum1 from the conditional sum adder with either manv or mann to form the final mantissa. Table III shows the truth table to the selection logic. The key to the table is the expression for the *Overflow* signal, shown in Equation 14. The first expression refers to the case where the MSB is set as a result of a carry within the addition of the 51 bits without a carry into bit 54. The second expression refers to the case where the MSB is set due to some carry into bit 54 in the non-overflow case. This carry may be due to rounding itself, or the case of overflow after rounding.

$$Overflow = \text{sum0}[105] + (\text{c54\_n} \cdot \text{sum1}[105]) \quad (14)$$

TABLE III
SELECTION LOGIC

Overflow	*c54_n*	*c54_v*	*Select*
0	0	x	sum0[104:54], mann[1:0]
0	1	x	sum1[104:54], mann[1:0]
1	x	0	sum0[105:54], manv[0]
1	x	1	sum1[105:54], manv[0]

4) Shared Hardware with Divide and Square Root

In our implementation of the floating point unit, the rounding for multiplication is similar to that needed for division and square root. In the interest of saving area, multiplication, division, and square root all share the same rounding hardware. Only additional muxing between the multiply, divide, or square root results is required before the inputs to the block shown in Figure 9.

One difference, however, between multiplication, division, and square root is the handling of the mantissas overflow. In

multiplication, the incremented exponent is used if an overflow occurs. In division and square root, however, the decimal point is taken to be immediately to the right of the MSB. Therefore, if the mantissa's MSB is zero, then the decremented exponent is selected. Table IV shows how the exponent is selected for multiplication, division, and square root.

TABLE IV
EXPONENT SELECTION FOR MULTIPLY AND DIVIDE

Mantissa	*Multiply*	*Divide/Sqrt*
Overflow	$e_r + 1$	e_r
Non-overflow	e_r	$e_r - 1$

IV. CONCLUSION

We have presented the design and implementation of a high-speed floating point multiplier. Partial subnormal support has been implemented with minimal addition to hardware and penalty on performance. Delay matching techniques were used in the multiplier tree and in the final addition stages. The rounding hardware is shared with the divide and square root units.

V. ACKNOWLEDGEMENT

The authors would like to thank Marc Tremblay, Arjun Prabhu, and the Program Committee for their valuable suggestions, and Nasima Parveen for her contributions in verification.

REFERENCES

[1] M. Mehta, et al, "High-speed multiplier design using multi-input counter and compressor circuits," *Proceedings 10th Symposium on Computer Arithmetic*, pp. 43-50, 1991.

[2] M. Nagamatsu, et al, "A 15 ns 32x32 bit cmos multiplier with an improved parallel structure," *Custom Integrated Circuits Conference*, pp. 10.3.1-10.3.4, 1989

[3] L. P. Rubinfield, "A proof of the modified Booth's algorithm for multiplication," *IEEE Trans. Comput.*, pp. 1014-1015, Oct. 1975

[4] C. S. Wallace, "A suggestion for parallel multipliers," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 14-17, Feb. 1964

[5] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349-356, 1965

[6] L. Dadda, "On parallel digital multipliers," *Alta Frequenza*, vol. 45, pp. 574-580, 1976.

[7] A. D. Booth, "A signed multiplication technique," *Quarterly J. Mechan. Appl. Math*, vol. 4, pt. 2, pp. 236-240, 1951

[8] D. Zuras and W. McAllister, "Balanced delay trees and combinatorial division in VLSI," *IEEE J. Solid-State Circuits*, vol. SC-21, no. 5, pp.814-819, Oct. 1986

[9] Email correspondence with Vojin Oklobdzija, 1993.

[10] D. T. Shen and A. Weinberger, "4-2 carry-save adder implementation using send circuits", *IBM Technical Disclosure Bulletin*, vol. 20, no. 9, Feb 1978.

[11] J. Sklansky, "Conditional sum addition logic", *Trans. IRE*, vol. EC-9, no. 2, pp. 226-230, June 1960.

[12] M. Santoro and M. Horowitz, "A pipelined 64x64b iterative array multiplier", *IEEE Int. Solid-State Circuits* conf., pp.35-36, Feb. 1988.

[13] M. Santoro, G. Bewick, and M. Horowitz, "Rounding algorithms for IEEE multipliers", *IEEE 9th Symposium on Computer Arithmetic* proceedings, pp. 176-183, Sept. 1989.

[14] V. Peng, S. Sanudrala, M. Gavrielov, "On the implementation of shifters, multipliers, and dividers in VLSI floating point units", *IEEE 8th Symposium on Computer Arithmetic* proceedings, pp. 95-102, May 1987.