# Multiplier Policies For Digital Signal Processing

*Gin-Kou Ma and Fred J. Taylor*

*Abstract* —**The successful design of digital signal processing (DSP) systems and subsystems is often predicated on realizing fast multiplication in digital hardware. The many options currently available to DSP system designers offer a wide range of speed/complexity tradeoffs. These choices embrace both traditional methods and those only recently reported. The purpose of this tutorial is to provide the reader with a broad perspective of this important field and the pedagogy needed to understand the basic principles of digital multiplication. Topics relating to fast multiplication and scaling are subdivided into major groups which include stand-alone fixed-point multipliers, cellular arrays, memory intensive policies, homomorphic systems, and modular arithmetic. The presented theory already appears in many manifestations, which range from simple logic devices to custom VLSI/VHSIC embodiments. Based on a familiarity with these methods and concepts, the DSP design engineer/scientist should be better able to understand DSP hardware design issues.**

## I. INTRODUCTION

DSP, whether relating to filters or transforms, is an arithmetic intensive study with the predominant operation being multiply/accumulate. As a result, the key to performance is translating the basic algebraic procedures of addition and multiplication into fast, compact digital operations. To design high performance hardware multiply and divide units, these concepts must be understood at the system level. In this tutorial, the design of a host of multiplier strategies are studied in the context of their algorithms and architectures. Both traditional and non-traditional multipliers are presented and compared. As a result, DSP engineers and scientists can make more efficient use of available computational hardware to achieve more advantageous cost/performance metrics as well as to design or specify custom multiplier units if required.

The hardware/software/algorithm synergism problem is an area of intense study by both the industrial and the academic communities. The importance of this claim is well illustrated in DSP chips which emphasize relatively fast fixed-point multipliers in their RISC (Reduced Instruction Set Computer) instruction sets. The TMS 32020, for example, incorporated a 16 × 16-bit 200 ns multiplier which occupied about 40% of the chip area. Newer DSP chips belonging to the third generation, such as the TI 32030 and AT&T DSP32, have introduced architectural improvements which again have the fast multiplier as their cornerstone.

Another recent technological innovation is the dedicated multiply, multiply/accumulate, and numeric processor chip. They appear in both fixed-point and floating-point forms and are summarized in Table 1. These chips offer the designer a wide range of cost/speed/power tradeoffs which can be integrated into a wide variety of DSP designs. A variation on this theme is the arithmetic coprocessor chip (e.g., INTEL 8087/80287), which can perform higher order algebraic tasks (e.g., exponentiate, transcendentals, etc.) assigned to them by CPU (Central Processing Unit) chip more efficiently than the CPU itself. When cost is the overriding objective, arithmetic can be performed by the native CPU in software at the expense of throughput. If the other extreme is the objective (i.e., speed without regard to cost or power), highly pipelined custom or semicustom CMOS, bipolar, or ECL designs can be considered using MSI, LSI, or VLSI integration.

## II. TRADITIONAL METHODS

### *II.1. Shift-Add Architectures*

In its most primitive form, multiplication can be reduced to a set of interleaved additions using a full-adder as the principal computational tool. For a system using n-bit sign-magnitude representation, multiplication can be performed by inspecting successive bits of the multiplier (least significant bit first) one at a time, determining whether or not to add the multiplicand (called a partial product), weighting the result in accordance with the multiplier bit location (i.e., $2^i$ for $i$th bit position) by using a binary shift register, and sending the result to an accu-

Table 1.

FLOATING-POINT CHIPS

| Type | 32b<br>M-FLOPS | 64b<br>M-FLOPS | Power<br>W |
|---|---|---|---|
| AMD | 8 | N/A | 7.5 |
| Analog Dev. | 10 | 2.5 | 0.4 |
| Weitek | 4 | 2 | 2.0 |
| | 8 | 8 | 2.0 |
| | 10 | N/A | 2.0 |
| Bipolar Integration Tech. | N/A | 45 | 5.5 |
| IDT | N/A | 10 | 0.75 |
| TI | N/A | 14.7 | 1.0 |
| TRW | 10 | N/A | 0.21 |

FIXED-POINT CHIPS

| Type | 12 × 12 MUL | 12 × 12 MAC | 16 × 15 MUL | 16 × 16 MAC | 24 × 24 MUL |
|---|---|---|---|---|---|
| Analog Dev. | 110n | 130n | 75n | 85n | 200n |
| TRW | N/A | 135n | 45n | 50n | N/A |
| AMD | N/A | N/A | 90n | N/A | N/A |
| Logic Dev. | 80n | N/A | 45n | 45n | N/A |
| Weitek | N/A | N/A | 55n | 75n | N/A |
| IDT | 30n | 30n | 35n | 35n | N/A |
| CYPRESS | N/A | N/A | 45n | N/A | N/A |

mulator. Such an indirect "shift-add" multiply using three $n$-bit registers is shown in Figure 1. The registers are denoted Accumulator (AC), Multiplier Register (MR), and Auxiliary Register (AX). The concatenation of AC with MR, denoted AC & MR, provides for a double precision $2n$-bit space in which to store the final full precision product. The content of the AC is initialized to zero while the multiplicand X and the multiplier $Y$ are loaded into the registers AX and MR respectively. Two separate flip-flops, denoted $X_S$ and $Y_S$, are used to store the signs of $X$ and $Y$ (That is:

$$\left.\begin{array}{r} X_S \leftarrow 0 \\ Y_S \leftarrow 0 \end{array}\right\} \text{unsigned operation}$$

$$\left.\begin{array}{r} X_S \leftarrow X_{n-1} \text{ and } X_{n-1} \leftarrow 0 \\ Y_S \leftarrow Y_{n-1} \text{ and } Y_{n-1} \leftarrow 0 \end{array}\right\} \text{sign-magnitude operation}$$

where $K_i$ is the $i$th bit of $K$). In addition, negative input operands ($X$ and $Y$) need to be converted to sign-magnitude form before entering the multiply cycle (That is:

$$\left.\begin{array}{ll} \mathrm{AX} \leftarrow \overline{\mathrm{AX}} + 1 & \text{(2's Complement)} \\ \mathrm{AX} \leftarrow \overline{\mathrm{AX}} & \text{(1's Complement)} \end{array}\right\} \text{if } X_S = 1$$

$$\left.\begin{array}{ll} \mathrm{MR} \leftarrow \overline{\mathrm{MR}} + 1 & \text{(2's Complement)} \\ \mathrm{MR} \leftarrow \overline{\mathrm{MR}} & \text{(1's Complement)} \end{array}\right\} \text{if } Y_S = 1$$

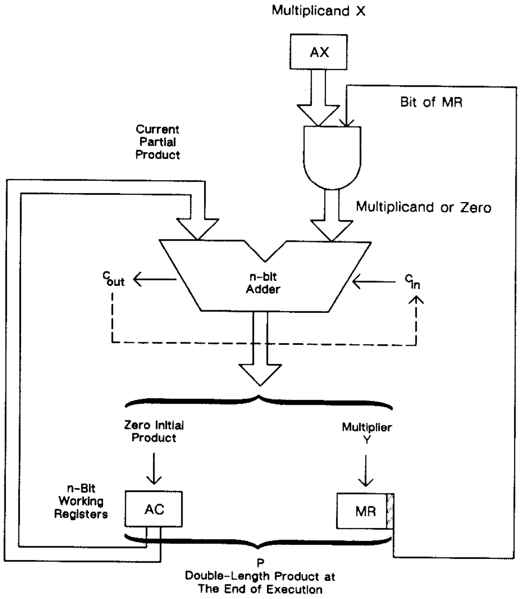
where $\overline{K}$ is the complement of $K$). An $n$-bit parallel adder is required to accumulate the partial products. The adder may range from a low complexity ripple adder to a fast adder (carry lookahead) configuration at the expense of added hardware and cost. The resulting full precision $2n$-bit product $P$ appears in the register pair AC & MR. The leading bit in AC, namely $AC_{n-1}$ ($= X_S \oplus Y_S$), carries the sign of the resulting product. A Sequence Counter (SC) keeps track of the number of add-shifts completed and recognizes the completion of the multiplication cycle. When 1's or 2's complement data is considered, the resulting sign-magnitude product is returned to the original format. If the product is negative (i.e., $X_S \oplus Y_S = 1$), then $P = \overline{P} + 1$ (2's) or $P = \overline{P}$ (1's) represents a post complement operation. The time sequence and spatial (bit) location of shift-add multiplication is summarized in Figure 1.b in terms of literals and in Figure 1.c where the same data is interpreted in what is called "dot-position form."

A variation on this theme uses a mix of full-adders and subtractors as shown in Figure 1.d. This unit can process 2's complement data directly. The inclusion of subtractor units allows the partial products generated by sign bits to be correctly blended into the accumulator. To understand its operation, recall that the procedure used to extend the wordlength of a 2's complement word is to copy the sign bit into the immediate left position. For example:

$$X = -8_{10} \rightarrow 1000_{2's} \text{ (4 bits)} \rightarrow \boxed{1}1000 \text{ (5 bits)}$$

$$X = 7_{10} \rightarrow 0111_{2's} \text{ (4 bits)} \rightarrow \boxed{0}0111 \text{ (5 bits)}$$

This wordwidth technique is needed to accommodate the partial sum expansion as indicated in the Figure 1.d. Thus, the product can be generated directly by using the



Unsigned Operation: $X_s \leftarrow 0,\ Y_s \leftarrow 0$

Sign-Magnitude: $X_s \leftarrow X_{n-1},\ Y_s \leftarrow Y_{n-1},\ X_{n-1} \leftarrow 0,\ Y_{n-1} \leftarrow 0$

1's or 2's Complement Operation:
$$\begin{cases} X_s \leftarrow X_{n-1},\ Y_s \leftarrow Y_{n-1} & & \\ AX = \overline{AX} + 1 & (2\text{'s}) & \\ AX = \overline{AX} & (1\text{'s}) & \text{if } X_s = 1 \\ MR = \overline{MR} + 1 & (2\text{'s}) & \\ MR = \overline{MR} & (1\text{'s}) & \text{if } Y_s = 1 \\ P = \overline{P} + 1 & (2\text{'s}) & \\ P = \overline{P} & (1\text{'s}) & \text{if } X_s \oplus Y_s = 1 \end{cases}$$

Figure 1. (a) The basic hardware for add-shift multiplication with uniform single right shift (AC-MR) per machine cycle.

$$\begin{array}{rc*{16}{c}l}
X & = & & & & & & & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & \leftarrow \text{MULTIPLICAND} \\
Y & = & & & & & & & & & Y_7 & Y_6 & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 & Y_0 & \leftarrow \text{MULTIPLIER} \\
\cline{11-18}
Y_0(X) & = & & & & & & & & & A_7 & A_6 & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 & \leftarrow \text{A PARTIAL PRODUCT} \\
2Y_1(X) & = & & & & & & & & B_7 & B_6 & B_5 & B_4 & B_3 & B_2 & B_1 & B_0 & & \\
4Y_2(X) & = & & & & & & & C_7 & C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0 & & & \\
8Y_3(X) & = & & & & & & D_7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0 & & & & \\
\cdot & & & & & & E_7 & E_6 & E_5 & E_4 & E_3 & E_2 & E_1 & E_0 & & & & & \\
\cdot & & & & & F_7 & F_6 & F_5 & F_4 & F_3 & F_2 & F_1 & F_0 & & & & & & \\
\cdot & & & & G_7 & G_6 & G_5 & G_4 & G_3 & G_2 & G_1 & G_0 & & & & & & & \\
 & & & H_7 & H_6 & H_5 & H_4 & H_3 & H_2 & H_1 & H_0 & & & & & & & & \\
\cline{3-18}
 & & P_{15} & P_{14} & P_{13} & P_{12} & P_{11} & P_{10} & P_9 & P_8 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 &
\end{array}$$

Figure 1. (b) Multiplying two 8-bit operands in eight partial products which are added to form a 16-bit final product.

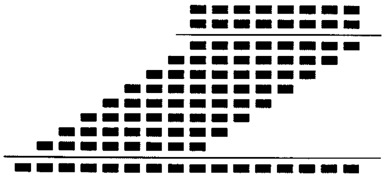


Figure 1. (c) Convenient dot representation of the same multiplication.

mix of full-adders and subtractors.

### *II.2. Booth's Algorithm and Modified Version [1] [2]*

It can be observed (Figure 1.b) that the shift-add operations associated with multiplier bits $Y_i = 0$ can be omitted (i.e., No Ops). Therefore, a potential speed-up of the multiplication process can be achieved if the multiplier is densely populated with zeros. Booth's algorithm exploits this observation by efficiently processing strings of zeros or ones. When they don't exist, attempts are made to synthesize a null string. The policy suggested in Booth's original algorithm was to skip over any contiguous string of all 1's or all 0's found in the multiplier. That is, rather than form a partial product for each multiplier bit, replace them with a string of No Ops. Observe first that a string of $n$ 1's represents the number $2^n - 1$ which is algebraically equivalent to a 1 followed by $n$ 0's less 1. In multiplication, this number is simply a shifting and subtraction operation. In principle, a Booth's decoder will scan a string of binary valued digits, say $Y_{n-1}, Y_{n-2}, \ldots, Y_2, Y_1, Y_0 (Y_0 = \text{LSB})$, search for strings of 0's or 1's, and implement the correct response.

Table 2. Input-Output Mapping for a Modified Booth Multiplier for Z = X * Y.

| $Y_{i-1}$ | $Y_i$ | $Y_{i+1}$ | Event |
|---|---|---|---|
| 0 | 0 | 0 | K ← K; K = Accumulator |
| 1 | 0 | 0 | K ← K + X (end of string) |
| 0 | 1 | 0 | K ← K + X |
| 1 | 1 | 0 | K ← K + 2K (end of string) |
| 0 | 0 | 1 | K ← K − 2X (beginning of string) |
| 1 | 0 | 1 | K ← K − X |
| 0 | 1 | 1 | K ← K − X (beginning of string) |
| 1 | 1 | 1 | K ← K |

The popular modified Booth's algorithm, given in Table 2, partitions the multiplier into 3-bit word substrings with adjacent groups sharing a common overlap bit. For an $n$-bit field, $\lceil n/2 \rceil$ discrete 3-bit "snapshots" will be taken and processed. The Booth database must first be initialized by affixing a "precondition" bit $Y_{-1}$ to the LSB end of the multiplier input data string. If $Y_{-1} = 0$, then $P = X*Y$ and if $Y_{-1} = 1$, then $P = X*Y + X$. To work with unsigned numbers, the $n$-bit multiplier must also be padded with zeros on the MSB side to fill out a data field extending from $Y_{-1}$ to $Y_m$ where $m = n$ if $n$ is even, $m = n + 1$ otherwise. If a 2's complement code is used, the sign bit is copied into the padded locations.

Initially, Booth's algorithm begins with the lowest order digits of the multiplier bit string $Y_1 Y_0 Y_{-1}$ ($Y_{-1}$ is artificially synthesized in this case). The next operation is defined by $Y_3 Y_2 Y_1$ ($Y_1$ is common to the current and prior case). Because of their relative position within the data field, the second action has 4 times the weight (significance) of that associated with $Y_1 Y_0 \cdot Y_{-1}$. Thus the table lookup di-

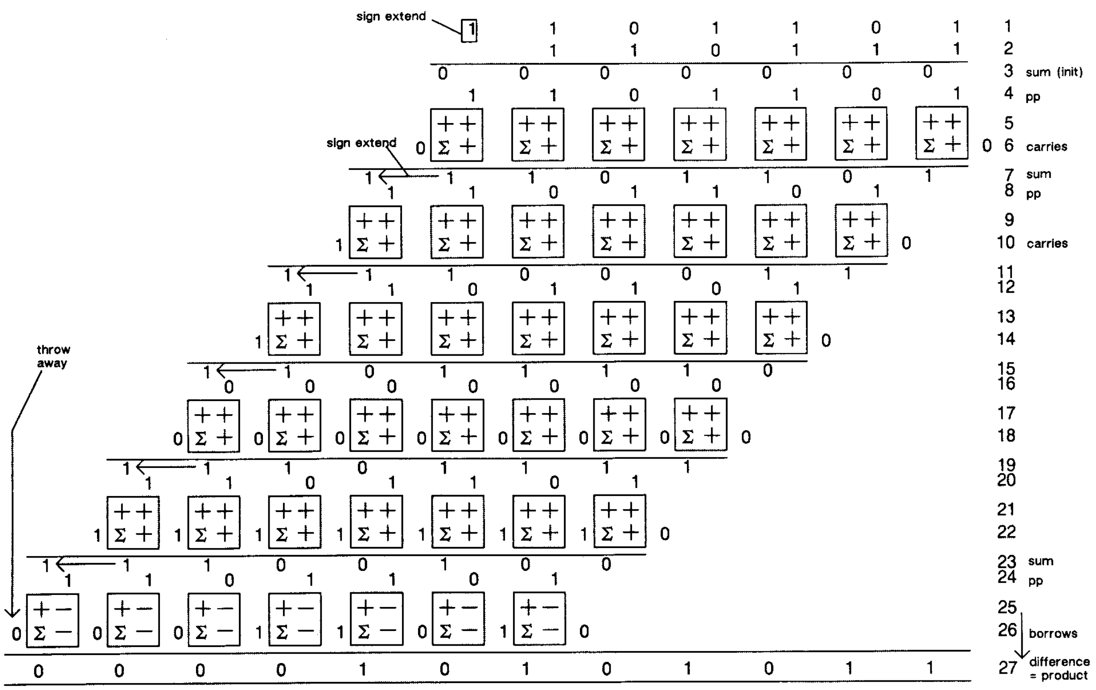


Figure 1. (d) The shift-add multiplier using a mix of full-adders and subtractors.

rective of $Y_3 Y_2 Y_1$ requires that the action specified in Table 2 be shifted by 2 bit locations before being sent to an accumulator (i.e., multiplied by 4) and so on. For example, suppose the multiplier with a padded $Y_{-1} = 0$ was 0110.0. The action taken is then $4{*}2x - 2x = 6x$. In summary, upon inspection of the Table 2, one notes that only one action (addition, No Op, or subtraction) is required for each three bits with overlap slice.

2's complement subtraction is implemented by first complementing the $X$ multiplicand and then adding 1 (with respect to the radix point) as a carry-in to the accumulator. In implementing this task, the binary digit $Y_{2i+1}$ can be used as a subtractor indicator. Of course, in 2's complement, the sign bit must be extended to the full width of the final result. For example, in Figure 2, if $X$ and $Y$ are 8-bit unsigned numbers, then the first partial product ($A_8 - A_0$) is determined by $Y_1 Y_0 0$. Since $\pm 2X$ is a possible action, $A_8$ may be affected and $A_9$ is the sign and extension. If a $\pm X$ action is determined, then $A_8$ is the sign and $A_9(= A_8)$ is the extension. The partial product ($E_7 - E_0$) is determined by $00Y_7$ which only generates positive, unshifted action ($+X$ or 0). If $X$ and $Y$ are 2's complement numbers, then $A_9(= A_8)$ is the sign and extension and no ($E_7 - E_0$) is required. The modified Booth's algorithm always generates $n/2$ independent partial products, whereas the Booth's algorithm generates a varying (at most $n/2$) number of partial products, depending on the bit pattern of the multiplier.

$$\begin{array}{l}
A_9\,A_9\,A_9\,A_9\,A_9\,A_9\,A_9\,A_8\,A_7\,A_6\,A_5\,A_4\,A_3\,A_2\,A_1\,A_0 \\
B_9\,B_9\,B_9\,B_9\,B_9\,B_8\,B_7\,B_6\,B_5\,B_4\,B_3\,B_2\,B_1\,B_0 \quad Y_1 \\
C_9\,C_9\,C_9\,C_8\,C_7\,C_6\,C_5\,C_4\,C_3\,C_2\,C_1\,C_0 \quad Y_3 \\
D_9\,D_8\,D_7\,D_6\,D_5\,D_4\,D_3\,D_2\,D_1\,D_0 \quad Y_5 \\
E_7\,E_6\,E_5\,E_4\,E_3\,E_2\,E_1\,E_0 \quad Y_7 \\
\hline
P_{15}P_{14}P_{13}P_{12}P_{11}P_{10}P_9\,P_8\,P_7\,P_6\,P_5\,P_4\,P_3\,P_2\,P_1\,P_0
\end{array}$$

Figure 2. Modified Booth's algorithm for 8 × 8 multiplication: five partial products are generated (if the representation is restricted to 2's complement, only four partial products are generated).

### *II.3. Wallace Trees [1] [2]*

Booth's algorithm belongs to a class of algorithms called "divide and conquer." Another divide and conquer algorithm is the Wallace tree. To accelerate a shift-add multiplier, it is desired to reduce $n$ partial products down to the final two partial products as fast as possible. The remaining two products can then be combined using a fast carry-lookahead adder or other options. Wallace tree elements can be used to support the partial product reduction. The classic 3-input Wallace tree element (denoted $W_3$) is a carry-save adder which accepts three-bit wide operands and exports a two-bit result (Figure 3.a), although larger Wallace tree primitives (e.g., 7 to 3) can also be used. The design of an 8 × 8 multiplier using $W_3$ elements is reported in Figure 3.b. Here, the partial products are sliced and added in a column-wise fashion. By recursively applying the Wallace tree to the columns of partial products, eight partial products can be reduced to two after four levels of recursive cycles through a network of $W_3$ elements.

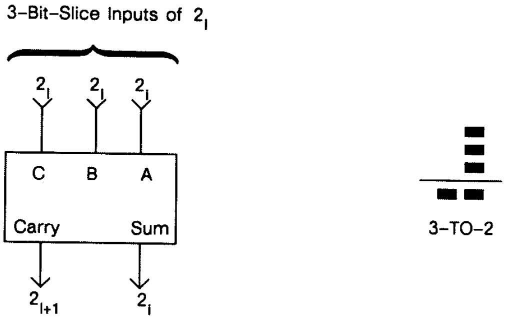


Figure 3. (a) A 3-bit-slice Wallace tree $W_3$.

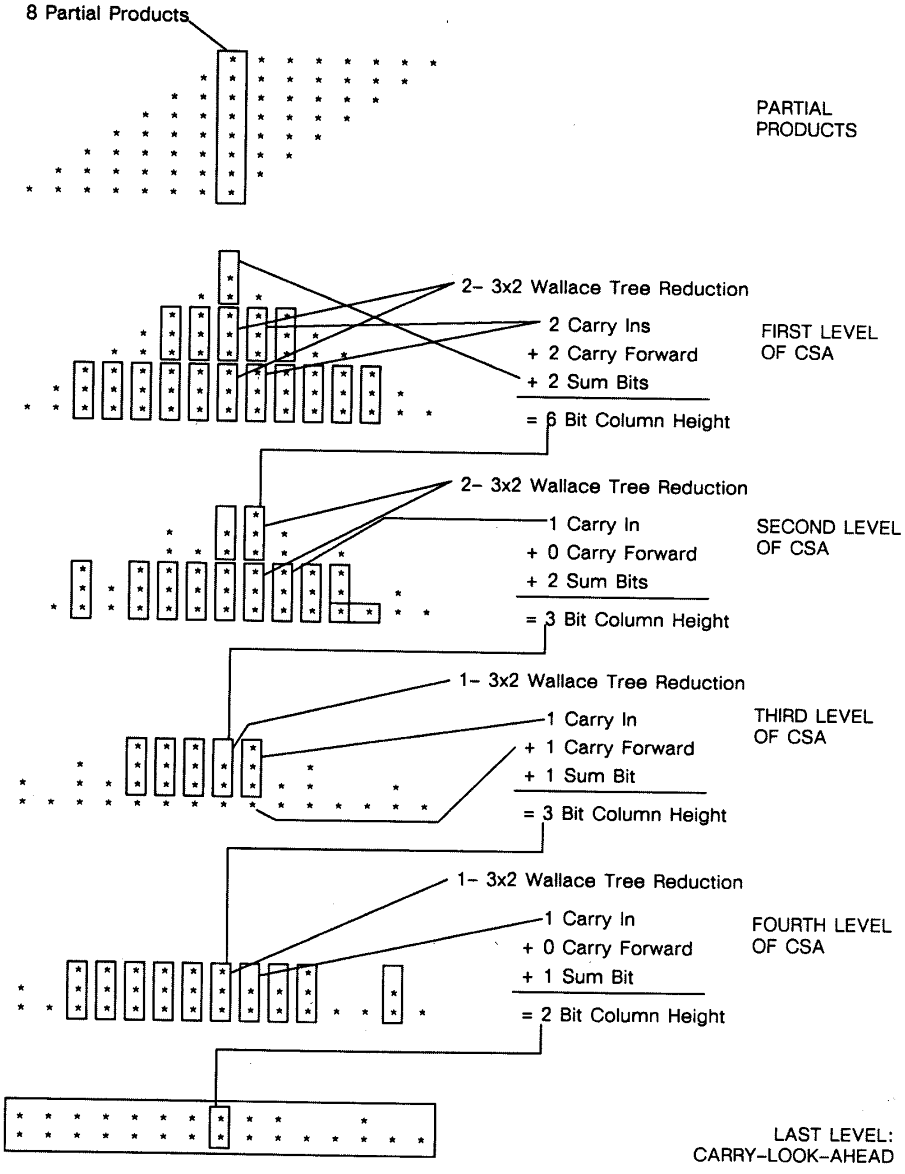


Figure 3. (b) Wallace tree reduction of 8 × 8 multiplication, using 3-to-2 carray-save adders (CSA).

## III. CELLULAR ARRAYS

### *III.1. Introduction*

The advent of MSI, LSI, and now VLSI has resulted in a profusion of monolithic multiplier devices (Table 1). If these units can operate as stand-alone multipliers, without the need to interconnect them to other multiplier chips, they are called Nonadditive Multiplier Modules (NMMs). However, it is often necessary to connect identical NMMs together to achieve a multiplier wordlength which exceeds the capability of a single module. Using multiple copies of an NMM, large wordlength multiplier arrays, called Cellular Arrays, can be designed. The objective of an effective cellular design is to recombine additively the subproduct outputs of the NMM cells in such a way so as to reconstruct the final product. For example, if a $2n$-bit word $W$ is represented as the weighted sum of two $n$-bit digits, such that $W = 2^n W_{HI} + W_{LO}$, then the product of two such numbers, say $X$ and $Y$, satisfies

$$\begin{aligned} P = A * B &= (X_H \& X_L) * (Y_H \& Y_L) \qquad (1) \\ &= 2^{2n}(X_H * Y_H) + 2^n(X_H * Y_L) \\ &\quad + 2^n(X_L * Y_H) + X_L * Y_L \\ &= 2^{2n}P_{HH} + 2^n(P_{HL} + P_{LH}) + P_{LL} \end{aligned}$$

where $A = X_H \& X_L$, $B = Y_H \& Y_L$, and $P_{ij} \in [0, 2^n - 2]$. In the above equation, the ampersand "&" refers to concatenation of two $n$-bit words. Therefore, the resulting product $P$ is the sum of four $2n$-bit subproducts with the assigned weights.

### *III.2. NMM Wallace Arrays*

The Wallace tree introduced an additional method of efficiently reducing many partial products down to a final two. The same technology can be used to combine NMM subproducts. Refer to Figure 4 where 4-bit wide NMMs are used as the basic cells. The unshaded area represents the design of an 8 × 8 cellular multiplier using $W_3$ Wallace trees to combine the 4 × 4 subproducts. The shaded area suggests the additional hardware (in terms of NMMs and Wallace trees) required to extend the multiplier beyond an 8 × 8 wordwidth.

### *III.3. Pezaris Multiplier Array*

A multiplication procedure is called an indirect 2's complement processor if negative numbers must first be converted to positive numbers before being presented to the multiplier unit. The desirability of direct 2's complement algorithms is obviously apparent. One algorithm which exhibits this property is due to Pezaris. To account for the presence of sign bit information, dispersed through the partial products, the special computational units displayed in Figure 5.a are used. An $n \times n$ Pezaris multiplier [2] requires $(n - 2)^2$ Type 0, $(n - 2)$ Type 1, $(2n - 3)$ Type 2, and (one) Type 3 full adders. A total of $n(n - 1)$ full adders are interconnected. Consider the 4 × 4 multiplier shown in Figure 5.b. The operations are illustrated by the summand matrix and a numerical example is shown in Figure 5.c (term $X_iY_j$ is called a summand, where $X_i$ is the ith bit of $X$ and $Y_j$ is the $j$th bit of $Y$). Here $x_3$ and $\bar{y}_3$ are negatively weighted signs and are marked by "¯". We also use "¯" to denote a negative summand term such as $\overline{X_iY_j}$. For example, the Pezaris product of the 4-bit 2's complement versions of ±5 would appear as:

$$\begin{array}{lccccccccl}
 & & & & \bar{0} & 1 & 0 & 1 & & = +5 \\
*) & & & & \bar{1} & 0 & 1 & 1 & & = -5 \\
\hline
 & & & & \bar{0} & 1 & 0 & 1 & & \\
 & & & \bar{0} & 1 & 0 & 1 & & & \\
 & & \bar{0} & 0 & 0 & 0 & & & & \\
 & 0 & \bar{1} & \bar{0} & \bar{1} & & & & & \\
\hline
 & 0 & \bar{1} & 0 & 0 & 1 & 1 & 1 & & \\
= & \bar{1} & 1 & 0 & 0 & 1 & 1 & 1 & & = -25 \\
 & | & | & & & & & & & 
\end{array}$$

Extended Sign Sign

### *III.4. Baugh-Wooley Multiplier*

The Baugh-Wooley array shares a degree of similarity with the previously reported Pezaris array. The major difference is that this architecture is designed using only positive (Type 0) summands. The Baugh-Wooley algorithm is architected as a mix of uncomplemented and complemented bit-wise partial products as suggested in Figure 6.

## IV. SPECIALIZED VLSI ARRAYS

### *IV.1. Introduction*

The impact of VLSI is increasing in the design of high performance arithmetic processing machines. VLSI has been successfully used to fabricate many of the general purpose multiply or multiply/accumulate units reported in Table 1. However, it is in the area of custom VLSI DSP chip design that significant breakthroughs may arise. Some, as in the case of DSP chips, were suggested earlier. Others are being designed to meet specific needs or serve special applications. Several of the more popular options are reported below.

### *IV.2. Systolic Array Multiplier [3] [12]*

Like the heart, systolic arrays [23] pass data from one processor to its neighbor processors in a regular, rhythmic pattern. The systolic arrays communicate to the external world through a limited number of input/output ports located at the periphery of the array. Systolic arrays also attempt to minimize computational propagation delay between processors by arranging them in a grid-like manner with highly localized communications. It is the potential speed advantage that makes systolic arrays currently popular among VLSI/DSP designers.

Consider, as an example, the convolution operation. This important DSP task is also closely related to vector-matrix multiplications. Define partial products $p_i$ to be

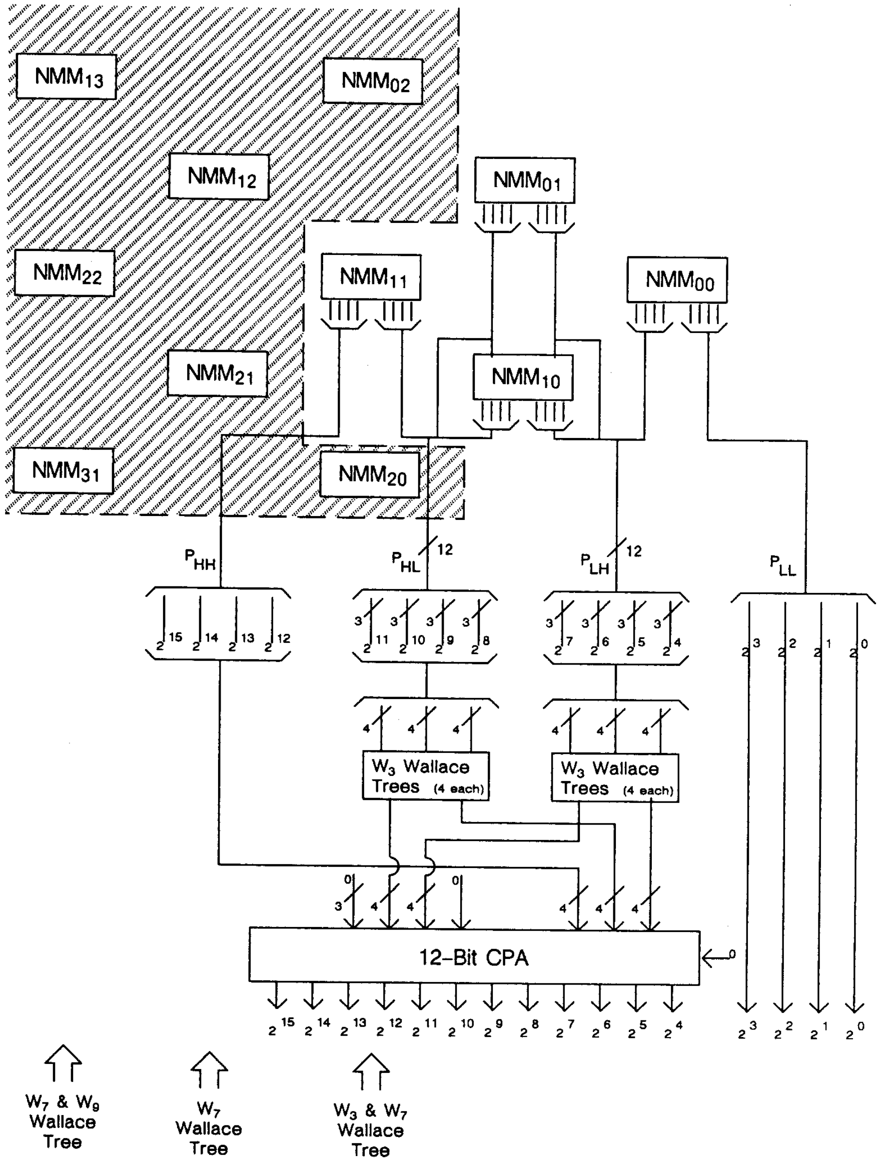


Figure 4. An 8 × 8 array multiplier build with 4 × 4 nonadditive multiply modulus (NMMs), Wallace trees, and carry propagate adder (CPA).

$$p_i = \sum_{j=0}^{n-1} x_j y_{i-j};\ p_0 = x_0 y_0,\ p_1 = x_1 y_1 \times x_0 y_1 \qquad (2)$$

and so on. Using VLSI technology, one can design a dedicated convolution system as a linear array of primitive multiply/accumulate processors. In order to limit the number of off-array I/O ports, only the processors at the edges, or array boundaries, will accept or export the $x$'s, $y$'s, or $p$'s. Since all the $x$'s cannot be read at once, one must read them as a sequentially accessed list. This is also the case for the $y$'s. Finally, the $p$'s are similarly initialized and exported from the array boundary.

There may exist a number systolic implementations of a given algorithm. In one embodiment, for example, the streams of $x$'s and $p$'s travel in alternate directions to neighboring processors. The $p$'s, which are initialized to zero, travel from the right. As they travel, they meet pairs of $x$'s and $y$'s which are producted and then added (accumulated) to the arriving $p$'s. The pattern we wish to achieve is shown in Figure 7. In each case, as shown in Figure 7.a, the actual array of processors extends only from the center to the right. That is, the $x$ values shown in the left of the center represent values that will be read in at future beats. Similarly, the $p$'s to the left of center

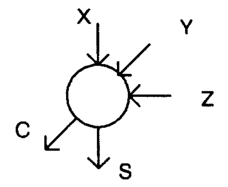
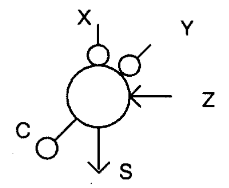
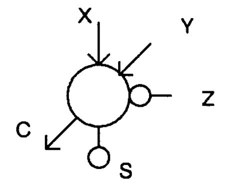
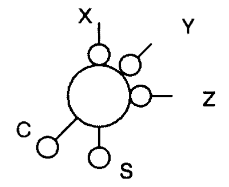
| TYPE | LOGIC SYMBOL | OPERATION | TYPE | LOGIC SYMBOL | OPERATION |
|---|---|---|---|---|---|
| TYPE 0<br>FULL<br>ADDER |  | X<br>Y<br>+) Z<br>C S | TYPE 2<br>FULL<br>ADDER |  | -X<br>-Y<br>+) Z<br>(-C)S |
| TYPE 1<br>FULL<br>ADDER |  | X<br>Y<br>+) -Z<br>C(-S) | TYPE 3<br>FULL<br>ADDER |  | -X<br>-Y<br>+) -Z<br>(-C)(-S) |

Figure 5. (a) Names and logic symbols of four types of generalized full adders [2].

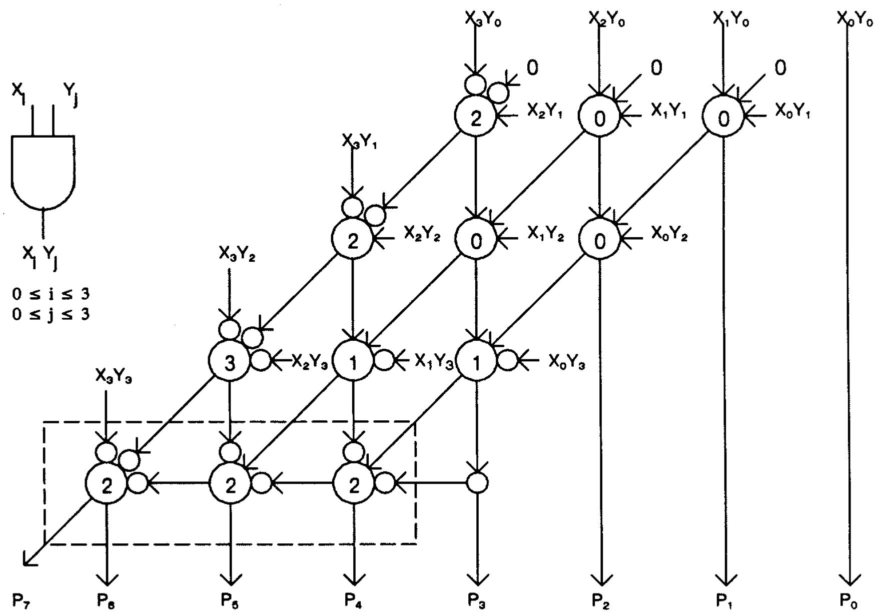


Figure 5. (b) The schematic circuit diagram of a 4 × 4 Pezaris array multiplier.

$$\begin{array}{cccccccc}
 & & & \overline{X_3} & X_2 & X_1 & X_0 & = A \\
 & & & \overline{Y_3} & Y_2 & Y_1 & Y_0 & = B \\
\hline
 & & & \overline{X_3Y_0} & X_2Y_0 & X_1Y_0 & X_0Y_0 & \\
 & & \overline{X_3Y_1} & X_2Y_1 & X_1Y_1 & X_0Y_1 & & \\
 & \overline{X_3Y_2} & X_2Y_2 & X_1Y_2 & X_0Y_2 & & & \\
X_3Y_3 & \overline{X_2Y_3} & \overline{X_1Y_3} & \overline{X_0Y_3} & & & & \\
\hline
P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 & = P
\end{array}$$

Figure 5. (c) The multiplication of two negatively signed 2's complement numbers.

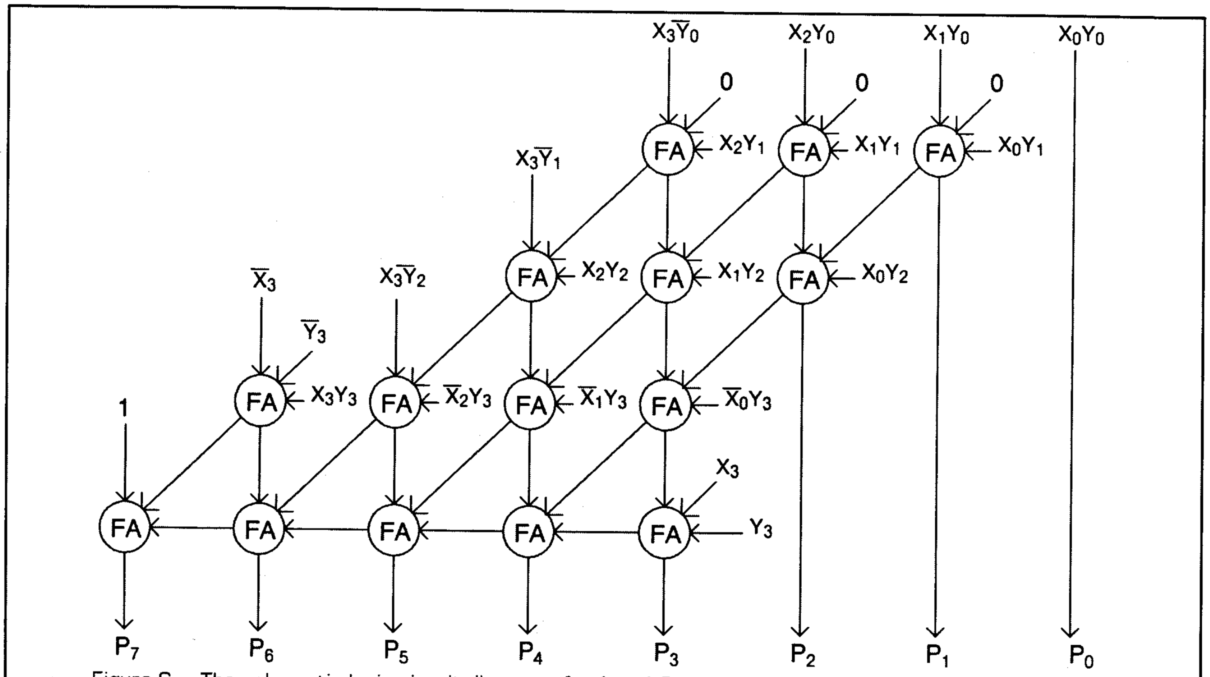


Figure 6. The schematic logic circuit diagram of a 4 × 4 Baugh-Wooley 2's complement array muliplier.

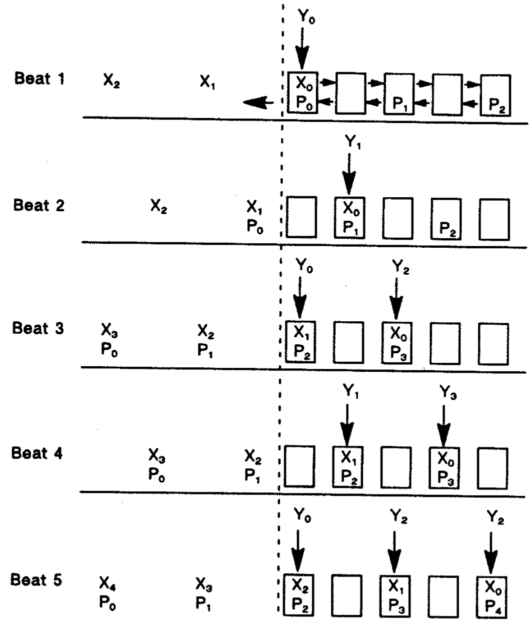
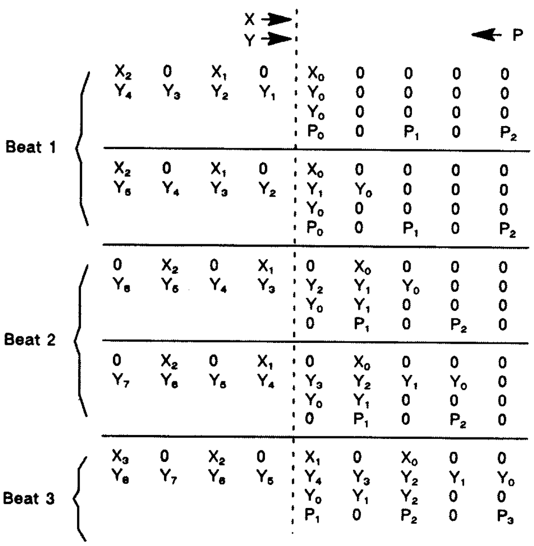


Figure 7. (a) Beats of a systolic convolution algorithm.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Beat 1 | $X_2$ | 0 | $X_1$ | 0 | $X_0$ | 0 | 0 | 0 | 0 |
| | $Y_4$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ | 0 | 0 | 0 | 0 |
| | | | | | $Y_0$ | 0 | 0 | 0 | 0 |
| | | | | | $P_0$ | 0 | $P_1$ | 0 | $P_2$ |
| | $X_2$ | 0 | $X_1$ | 0 | $X_0$ | 0 | 0 | 0 | 0 |
| | $Y_5$ | $Y_4$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ | 0 | 0 | 0 |
| | | | | | $Y_0$ | 0 | 0 | 0 | 0 |
| | | | | | $P_0$ | 0 | $P_1$ | 0 | $P_2$ |
| Beat 2 | 0 | $X_2$ | 0 | $X_1$ | 0 | $X_0$ | 0 | 0 | 0 |
| | $Y_6$ | $Y_5$ | $Y_4$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ | 0 | 0 |
| | | | | | $Y_0$ | $Y_1$ | 0 | 0 | 0 |
| | | | | | 0 | $P_1$ | 0 | $P_2$ | 0 |
| | 0 | $X_2$ | 0 | $X_1$ | 0 | $X_0$ | 0 | 0 | 0 |
| | $Y_7$ | $Y_6$ | $Y_5$ | $Y_4$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ | 0 |
| | | | | | $Y_0$ | $Y_1$ | 0 | 0 | 0 |
| | | | | | 0 | $P_1$ | 0 | $P_2$ | 0 |
| Beat 3 | $X_3$ | 0 | $X_2$ | 0 | $X_1$ | 0 | $X_0$ | 0 | 0 |
| | $Y_8$ | $Y_7$ | $Y_6$ | $Y_5$ | $Y_4$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
| | | | | | $Y_0$ | $Y_1$ | $Y_2$ | 0 | 0 |
| | | | | | $P_1$ | 0 | $P_2$ | 0 | $P_3$ |

Figure 7. (b) Details of systolic convolution.

represent values that have already been written out.

There are two ways to communicate the $y$'s to the required physical array location and have them arrive at the correct time. First, consider pipelining the $y$'s in from the right, with $y_i$ traveling along with $p_i$ (which is not read in, but is initialized to 0). When any $y_i$ meets its corresponding $x_0$, the $y_i$ freezes in place. Alternatively, as shown in Figure 7.b, the $y$'s could be read in from the left end along with the $x$'s. However, the $y$'s would travel twice as fast as the $x$'s. Finally, when a $y_i$ catches up with $x_0$, the

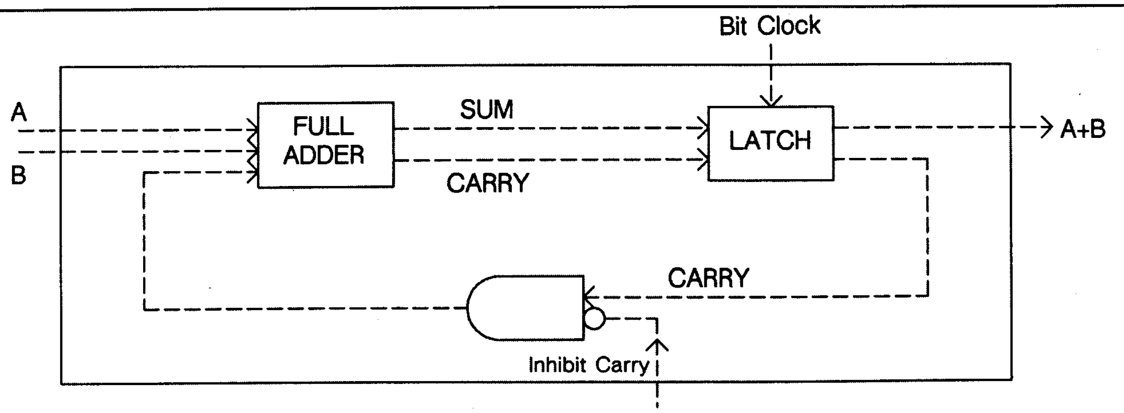


Figure 8. A bit-serial adder.

$y_i$ freezes.

The disadvantage of the first method is that $y$'s must travel from right to left through $n$ processors before they can be used. With the second method, the $p$'s are being produced every two beats. This does not mean that the first method's latency is longer than the second. In fact, the latter requires more beats to complete the task than the former. However, it should be noted that the output of the second method can be piped to another systolic process with only a one-beat delay, rather than an $n$-beat delay. This might be significant if one were multiplying together many variables where the $p$'s from one product become the $x$'s for the next product and so on.

### *IV.3. Bit-Serial Multiplier [4] [5]*

Bit-serial architectures require a communication policy that is also well suited to VLSI implementation. In a bit-serial design, information is transmitted sequentially along bit wide channels bit-by-bit. This is in sharp contrast to conventional architectures, which communicate as word-wide transactions. As a result, highly functional VLSI designs can be realized. Bit-serial networks can be easily routed and interconnected without the problems of bit-parallel busing, since all signals enter and leave the chip via single pin ports. The single pin communication mitigates the I/O pin limitation of VLSI chips and makes multi-function chip design (automatic system synthesis) possible. Furthermore, if the bit-serial process is highly pipelined, high throughputs can also be realized.

The key bit-serial computational element is the serial adder shown in Figure 8. This device is similar to a carry-save adder. With this device, a number of bit-serial arithmetic units can be designed, including a multiplier. In fact, one of the primary uses of the bit-serial policy is the design of high-throughput, area efficient multipliers. Based on a serial adder primitive, a shift-add multiplier can be readily designed. Because of the limited number of I/O pads and area required to build a single multiplier cell, it can be seen that many multiplier cells can be integrated on a common chip and used to simultaneously process data on independent channels. Of course, other multiplier structures, such as the modified Booth's algorithm, can also be developed as bit-serial statements.

### *IV.4. Distributed Arithmetic [21]*

The previously referenced methods have multiplication as their common objective. Technically, multiplication is the algebraic combination of two variables using a set of consistent multiplication rules. A special case is called scaling which applies the same combining rules to a variable and a constant. Such operations are ubiquitous in DSP and are often found as "coefficient scaling" operations. For example, a simple finite impulse response (FIR) is given by

$$y(k) = \sum_{j=0}^{L-1} a_j x(k-j) \tag{3}$$

where $\{a_j\}$ are known a priori coefficients. Any of the previously referenced fixed-point multiplier schemes could be used to accept $a_j$ and $x(k - j)$ as input operands and export a partial product. If each multiply-add cycle requires $t_c$ units of time, the $y(k)$ can be produced in approximately $L * t_c$ time units. An alternative scheme, called distributed arithmetic (DA), has been proposed and based on interpreting (3) as a binary weighted summation. If $x(r)$ is an $n$-bit 2's complement word, then

$$x(r) = \sum_{m=0}^{n-2} 2^m x[r:m] - 2^{n-1} x[r:n-1] \tag{4}$$

where $x[r:v]$ is the $v$th bit of $x(r)$, $x[r:v] \in [0,1]$. Upon substitution, (3) becomes

$$y(k) = \sum_{m=0}^{n-2} 2_m \Phi(m) - 2^{n-1}\Phi(n-1;$$

$$\Phi(q) = \sum_{j=0}^{L-1} a_j x[k-j:q] \tag{5}$$

Observe that $\Phi(q)$ is a function of the known set $\{a_j\}$ and the $q$th common bits of $x(r)$, $r = 0, 1, \ldots, L - 1$. Referring to Figure 9, it can be seen that the $L$-bit data field, obtained by accessing the $q$th common bit of $x(k)$ through $x(k - L + 1)$, is used as an $L$-bit address which is pre-

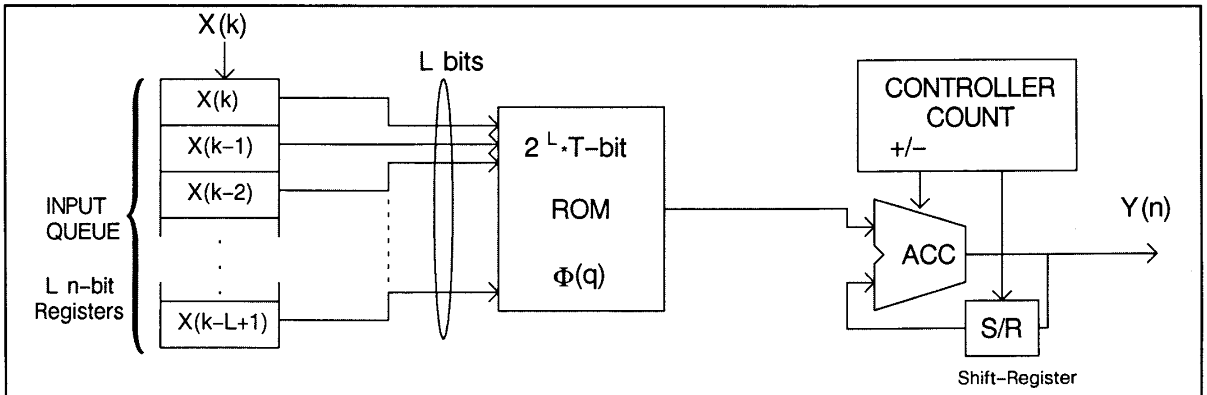


Figure 9. Distributed arithmetic implementation of sum-of-products.

sented to the read only memory (ROM) table $\Phi$. The ROM has been programmed with the $2^L$ precomputed values of $\Phi(m)$. Finally, scaling by $2^m$, as found in (3), can be implemented as a simple shift-add task. The result is that a filter cycle can now be completed in approximately $n$ memory cycle units. For example, if $L = 12$, $n = 16$, $t_{\text{MULT}} = 100$ ns and $t_{\text{MEM.CYC}} = 10$ ns, then a conventional multiplier filter cycle delay would be on the order of 1200 ns versus 160 ns for the distrubuted designs.

## V. NON-TRADITIONAL METHODS

### *V.1. Introduction*

In the previous section, popular fixed-point multiplier and scaling architectures were presented. They appeared in sign-magnitude, 1's, and 2's complement forms. In this section, alternative number systems will be presented. Each offers an unique solution to a particular DSP problem. Generically, they have been classified as:

- Canonic signed digit number systems
- Logarithmic number systems
- Residue number systems

We should review the most commonly used embodiments of these alternative systems in this section.

### *V.2. Canonic Signed Digit Number System [22]*

In the development of Booth's algorithm (Section II.2), the advantageous to of a multiplier densely populated with zero (No Ops) are noted. The designers of early vacuum tube technology computers were well aware of this, as well as of the ternary number system (i.e., $-1, 0, +1$). The ternary number system is the result of the fact that vacuum tube technology could implement $<, =, >$ tests with relative ease. Therefore, it was equally desirable to make a ternary coded word maximally dense in zeros or No Ops. The canonic signed digit number system achieved this goal and enjoyed a degree of popularity in the early 1950s. The canonical system is ternary. The digits $c_i$ of $x = c_{n-1}c_{n-2} \ldots c_0$ can be chosen so that $c_i * c_{i-1} = 0$. Since the product of two adjacent digits is zero, at least one digit must in fact be zero. Using this scheme, data words which have a maximal number of zeros embedded into their structure can be encoded. However, due to semiconductor advancements, other binary-valued numbering systems have rapidly replaced the canonical system. The canonical numbering system does, however, possess an unique feature which makes it attractive to certain DSP applications.

For a given $n$-digit representation of a number $x \in [-2^{n-1}, 2^{n-1})$, the canonical representation can be related to its 2's complement version through

$$x = -x_n 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \overset{\Delta}{=} \sum_{i=0}^{n-1} c_i 2^i \qquad (6)$$

where $x_i = (0, 1)$ and $c_i = (0, +1, -1)$. Whereas the 2's complement is binary, the $\{c_i\}$ are encoded as canonic digits. The utility of this is evidenced by the following simple example.

EXAMPLE: For $n = 3$, the integers $[-4, 3]$ are encoded as follows:

| 2's complement | | x | Canonical | |
|---|---|---|---|---|
| 011 | 12 zeros | 3 | 1 0 −1 | 15 zeros |
| 010 | | 2 | 0 1 0 | |
| 001 | | 1 | 0 0 1 | |
| 000 | | 0 | 0 0 0 | |
| 111 | | −1 | 0 0 −1 | |
| 110 | | −2 | 0 −1 0 | |
| 101 | | −3 | −1 0 1 | |
| 100 | | −4 | −1 0 0 | |

The probability that a canonical digit $c_i$ has a nonzero value is given by

$$P(|c_i| = 1) = 1/3 + (1/9n)[1 - (-1/2)^n] \qquad (7)$$

As $n$ becomes large, the probability tends to be 1/3. This compares to a value of 1/2 for a 2's complement code. Using the canonical structure, it can also be seen that the probability of a weighted digit being zero is approximately 2/3. The zero term will contribute nothing to the sum-of-products and may therefore be "by-passed" two thirds of the time. Thus, an overall speed-up can be achieved.

The re-coding of a 2's complement integer into a canonical form can be performed in the manner suggested by the data found in the Table 3 below.

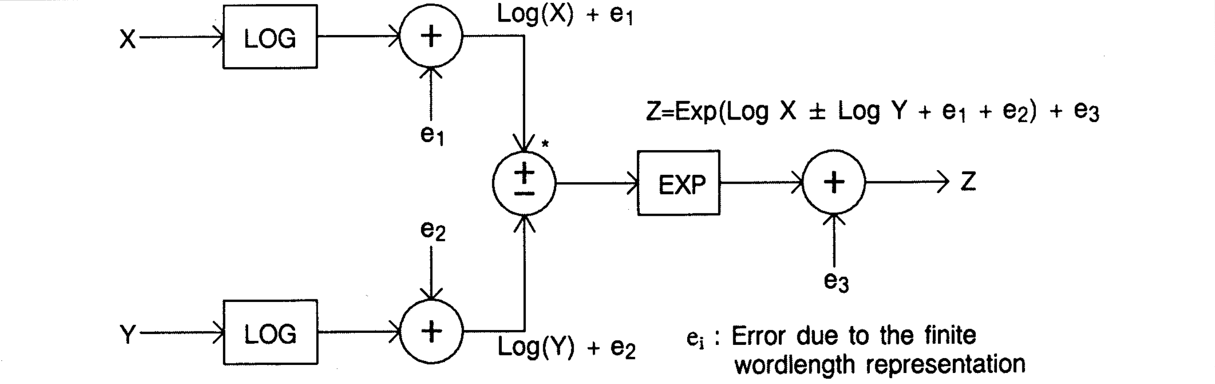

X
LOG
Log(X) + e1
e1
Z=Exp(Log X ± Log Y + e1 + e2) + e3
*
±
EXP
Z
e2
e3
Y
LOG
Log(Y) + e2
ei : Error due to the finite wordlength representation

Figure 10. Model for logarithmic multiplication/division showing all possible error sources. (*: addition for multiplication and subtraction for division).

Table 3. Conversion of 2's Complement to Canonical Digits.

| $c_i$ | $x_{i+1}$ | $x_i$ | $c_{i+1}$ | $C_i$ | Comment |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | $x_{i+1}$ and $x_i$ = 2's comple- |
| 0 | 0 | 1 | 0 | 1 | ment digits of x |
| 0 | 1 | 0 | 0 | 0 | $x_{n+1} = 0$ if $x \geq 0$ and $= 1$ if $x < 0$ |
| 0 | 1 | 1 | 1 | −1 | $c_i$ = ith carry digit of the |
| 1 | 0 | 0 | 0 | 1 | ith set |
| 1 | 0 | 1 | 1 | 0 | $c_0 = 0$ |
| 1 | 1 | 0 | 1 | −1 | $C_i$ = ith canonical |
| 1 | 1 | 1 | 1 | 0 | signed digit |

Such a scheme was used in the INTEL 2920 DSP chip. A supplied compiler coded the filter coefficients into a canonical form and then stored them in resident memory. The canonic digits were read digit-by-digit during run-time with the ±1 indicating an add or subtract directive to a shift-accumulator.

### *V.3. Logarithmic Number System [14–18]*

The logarithmic number system (LNS) has been studied in a DSP environment by numerous authors. In the LNS, the format of an LNS word is given by

$$X = \pm r^{\pm e_x} \tag{8}$$

where $r$ is the radix and $e_x$ is the signed fractional exponent. Figure 10 shows all possible error sources (resulting from the finite length effect of $e_x$) for multiple/divide operation in an LNS system. The LNS has a wide dynamic range and possesses a natural data companding capability. If one considers $X = 2^{e_x}$ and $Y = 2^{e_y}$ (assume a radix of $r = 2$), then arithmetic in this system is given by

$$\begin{aligned}
&\text{Multiply:} && Z = X * Y = 2^{e_x} * 2^{e_y} = 2^{e_x+e_y} = 2^{e_z}; \\
& && e_z = e_x + e_y \\
&\text{Divide:} && Z = X/Y = 2^{e_x}/2^{e_y} = 2^{e_x-e_y} = 2^{e_z}; \\
& && e_z = e_x - e_y = D \\
&\text{Add:} && Z = X + Y = 2^{e_x} + 2^{e_y} = 2^{e_x}(1 + 2^{e_x-e_y}) \\
& && = 2^{e_z}; \quad e_z = e_x + \log_2(1 + 2^{-D}) \\
&\text{Subtract:} && Z = X - Y = 2^{e_x} - 2^{e_y} = 2^{e_x}(1 - 2^{e_x-e_y}) \\
& && = 2^{e_z}; \quad e_z = e_x - \log_2(1 - 2^{-D}) \\
&\text{Square:} && Z = X^2 = 2^{2e_x}; \quad e_z = 2e_x \\
&\text{Square Root:} && Z = X^{1/2} = 2^{e_x/2}; \quad e_z = e_x/2
\end{aligned} \tag{9}$$

In the above list of operations, the outcome is denoted $Z$. However, $Z$ is not of primary interest. Rather, the exponent $e_z$ of the word $Z = r^{e_z}$ is the parameter of value.

Observe that the operations of multiply, divide, square, and square root only require exponent manipulation. These operations (i.e., addition or binary shift) are simple to implement in fast hardware. Addition and subtraction are, as one can see, a different story. Observe that the definition of $e_z = e_x + \log_2(1 \pm 2^{-D})$ is not derived but rather obtained as a memory lookup call. That is, $D$ is used as a table address which stores the precomputed value of $\log_2(1 \pm 2^{-D})$.

It is known that the LNS possesses a wider dynamic range and higher precision than fixed-point schemes. The precision and dynamic range capabilities of an LNS unit are essentially limited by the available address space of a high speed RAM or ROM. Various schemes have been proposed to overcome this problem. For example, a 20-bit $I^2L$ VLSI Arithmetic Processor based on the LNS has been reported [19]. It dissipates 390 mW and has computation delays of 40 ns for multiply/divide, 92 ns for add/substract, and 20 ns for square/square root.

### *V.4. Residue Number System [6–11]*

Recently, a tutorial paper on the subject of the Residue Number System (RNS) [7] established that this ancient branch of mathematics has enjoyed a renaissance over the last half-decade motivated by the promise of speed. The RNS, due to whose intrinsic parallel structure, can achieve very high throughput. Briefly, the RNS is an integer number system defined with respect to a set $P$ (called a moduli set) of relatively prime integers (called moduli). In particular, if $P = (p_1, \ldots, p_L)$ and $\text{GCD}(p_i, p_j) = 1$, then over the integer range $[0, M)$ ($[-M/2, M/2)$ for signed numbers), where $M = p_1 * p_2 * \ldots * p_L$, an integer $X$ is isomorphic to the $L$-tuple residue given by $X \to (x_1, x_2, \ldots, x_L)$, $x_i = X \bmod p_i$. If two integers $X$ and $Y$ have admissible RNS representations, and if they are algebraically com-

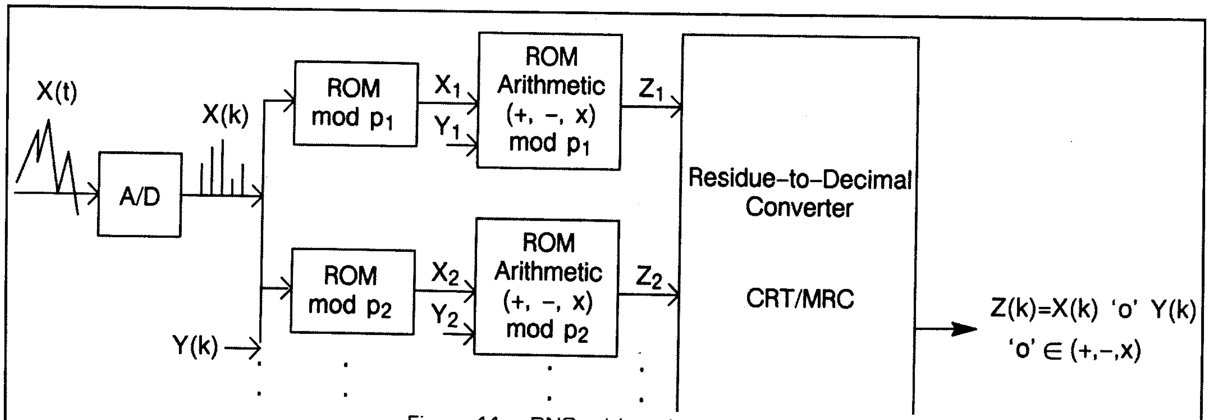

X(t)
A/D
X(k)
Y(k)
ROM mod p1
ROM mod p2
X1
Y1
X2
Y2
ROM Arithmetic (+, −, x) mod p1
ROM Arithmetic (+, −, x) mod p2
Z1
Z2
Residue-to-Decimal Converter
CRT/MRC
Z(k)=X(k) 'o' Y(k)
'o' ∈ (+,-,x)

Figure 11. RNS arithmetic processor.

bined under the arithmetic operation '$o$', where $0 = (+, -, *)$ (division is not closed in the RNS), then $Z = X \circ Y \rightarrow (x_i \circ y_i, \ldots, x_L \circ y_L) = (z_1, z_2, \ldots, z_L)$ if $Z$ belongs to the residue class $Z_M = \{0, 1, \ldots, M-1\}$. The remarkable feature of this statement is that in the RNS, $Z$ can be computed as $L$ concurrent operations without the need of producing carry digits or establishing a carry management policy! This pure parallelism is why the RNS theoretically has the ability to support ultra high-speed arithmetic.

The RNS has often been suggested as a promising medium in which to design very high-performance real-time (100 MHz) special purpose signal and image processing systems. The scenario would read as follows (see Figure 11):

- Convert a real signal into low wordlength (typically 6-12 bits) data using a "flash A/D converter" running at $10^6$ to $10^8$ Hz sample rate.
- Map the A/D converted $n$-bit sample into an RNS $L$-tuple using direct table lookups. That is, if $p_i < 2^{m_i}$, for $m_i$ sufficiently small (e.g., $m_i \leq$ 6-bits), then use a $2^n \times m_i$-bit ROM to map a signal $X$ ($n$-bits) into $X_i \equiv X \bmod p_i$ as a table lookup task. For large data wordlength, the integer-to-residue conversion can be implemented as a modular sum of $m_i$-bit reduced digits (i.e., $X_i = (X = \Sigma 2^{m_i} X_j) \bmod p_i$).

$$\left(X = \sum_{m=0}^{n-1} 2^m X_m\right) \bmod p_i \equiv \left[\left(\sum_{m=0}^{5} 2^m X_m \bmod p_i\right) + \cdots + \left(\sum_{m=n-6}^{n-1} 2^m X_m \bmod p_i\right)\right] \bmod p_i \tag{10}$$

- Perform arithmetic operations, restricted to add, subtract, or multiply, using direct table lookups. That is, send the two $m_i$ bit operands $X_i$ and $Y_i$ to a $2^{2m_i} \times m_i$-bit table which contains the precomputed value of $Z_i \equiv (X_i \circ Y_i) \bmod p_i$. Methodologies for keeping the ROM size down, and efficient algorithms and architectures to implement large moduli multipliers were shown in Section 3.2 of [24].
- Convert an RNS $L$-tuple back into an integer in order to service division related operations or to output results.

Historically, two routines have been used to convert an RNS $L$-tuple into an integer. One is known as the Chinese Remainder Theorem (CRT) and the other as the Mixed Radix Conversion (MRC) algorithm. They map a residue $L$-tuple into an integer by using a nested sum of modular partial products [25]. The bane of the RNS has traditionally been the inability to support efficient high-speed residue-to-decimal conversion (RDC). The RDC is a fundamentally important operation in magnitude comparison, sign detection, and overflow management operations and must be dealt with.

Fast parallel complex arithmetic is also known to be important to many numerically intensive computational tasks. To a computing machine, all arithmetic operations are real or are defined in terms of approximations to real numbers. Over the reals, the roots of $x^2 + 1 = 0$ do not exist and are therefore called imaginary numbers (i.e., $x = \pm i, i = \text{SQRT}(-1)$). The general rules for complex arithmetic, for a complex number $z = \{a + ib \mid a, b \in R\}$, are

$$\begin{aligned} &\text{Addition:} && (a + ib) \pm (c + id) = (a \pm c) + i(b \pm d) \\ &\text{Multiplication:} && (a + ib)(c + id) = (ac - bd) + i(b \pm d) \end{aligned} \tag{11}$$

As expected, complex addition (subtraction) requires two real additions (subtractions) while complex multiplication requires four real multiplications plus two real additions.

In another framework, namely a finite ring, one can again consider the roots of $x^2 \equiv -1 \bmod p$. If $x \notin Z_p$, then $x$ is said to be non-quadratic root or imaginary [13] [20]. Until recently, complex arithmetic in the RNS, called the Complex RNS (CRNS), was based on the classic rules for complex arithmetic, which was again four real multiplications plus two real additions per complex multiplication.

If one now considers the use of primes of the form $p = 4n + 1$, then the congruence $x^2 = -1 \bmod p$ has an integer solution $j$, called a quadratic root such that $j^2 = -1 \bmod p$ and $j \in Z_p$. That is, $j$ is now a real number (e.g., $p = 17, j = 4$ or 13 such that $4^2 = 16 = -1 \bmod 17, 13^2 =$

$169 = -1 \bmod 17$). It has been shown that there is an isomorphic mapping of a complex number $z = a + ib$ into two new two tuple integers $(c, c')$ under $(c, c') \leftarrow ((a + jb) \bmod p, (a - jb) \bmod p)$. It then follows that

$$\text{i.} \quad (c, c') + (d, d') = ((c + d) \bmod p, (c' + d') \bmod p)$$

$$\text{ii.} \quad (c, c') * (d, d') = ((c * d) \bmod p, (c' * d') \bmod p) \tag{12}$$

The importance of this result is that only two real multiplications and no additions are required to perform complex multiplication! It has also been reported that power operations, such as the ubiquitous magnitude squared mapping $z\bar{z} = |z|^2$ ($\bar{z}$ is the complex conjugate of $z$), can be now completed in one real multiplication versus two multiplications plus an addition as is currently required. This system is called the quadratic RNS or QRNS. Recently a single modulus QRNS (SM-QRNS) has been reported [9] and experimentally verified to offer about a 2:1 speedup advantage in performing complex multiplications over traditional designs.

## VI. SUMMARY

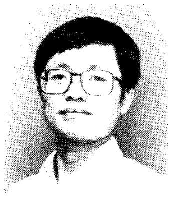
In this tutorial paper, both conventional and nonconventional methods of implementing multiplication have been presented. They represent a mix of speed/complexity tradeoffs. Some are based on traditional shift/add structures while others strive for greater mathematical sophistication. Booth's algorithm, for example, can halve the number of partial products required to complete a product. Cellular arrays can provide a mechanism for synthesizing long wordlength multipliers. The systolic and bit-sequential architectures may provide the solution to designs with limited I/O pads. To reduce multiplication delays, DA, RNS, and LNS methods may be considered (which are table lookup intensive with the calculation time almost equal to memory access time). Some techniques are currently supported by a host of commercially available hardware, others are not. All, however, offer the DSP system designer a rich mix of speed/complexity tradeoffs. It is the engineer's task to sort through this list and choose the one which is most applicable.

## ACKNOWLEDGMENTS

The authors wish to express their appreciation for the support of the ARO and Florida High-Technology Council for their support and to thank Monica Murphy, Mike Sousa, Glenn Zilnecker and George Papadourakis for their help in preparing this paper.

## REFERENCES

1. S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers,* CBS College Publishing, 1982.
2. K. Hwang, *Computer Arithmetic Principles, Architecture, and Design,* John Wiley & Sons, 1979.
3. J. D. Ullman, *Computational Aspects of VLSI,* Computer Science Press, Inc., 1984.
4. H. J. Sips, *Bit-Sequential Arithmetic for Parallel Processors,* IEEE Trans. on Computers, January 1984.
5. N. Kanopoulos, *A Bit:Serial Architecture for Digital Signal Processing,* IEEE Trans. on Circuits and Systems, March 1985.
6. J. H. McClellan and C. M. Rader, *Number Theory in Digital Signal Processing,* Prentice-Hall, Inc., 1979.
7. F. J. Taylor, *Residue Arithmetic: A Tutorial with Examples,* IEEE Trans. on Computers, May 1984.
8. M. C. Vanwormhoudt, *Structural Properties of Complex Residue Rings Applied To Number Theoretic Fourier Transforms,* IEEE Trans. on ASSP, February 1978.
9. F. J. Taylor, *Single Modulus Complex ALU,* IEEE Trans. on ASSP, October 1985.
10. F. J. Taylor, *A VLSI Residue Arithmetic Multiplier,* IEEE Trans. on Computers, June 1982.
11. F. J. Taylor and C. H. Huang, *An Autoscale Residue Multiplier,* IEEE Trans. on Computers, April 1982.
12. C. Mead and L. Conway, *Introduction to VLSI Systems,* Addison-Wesley Publishing, Inc., 1980.
13. I. N. Herstern, *Topics in Algebra,* 2nd Ed., Southwest Book Co., 1975.
14. N. G. Kingsburg and P. J. W. Rayner, *Digital Filtering using Logarithmic Arithmetic,* Electron. Letts., January 28, 1971.
15. F. J. Taylor, *An Extended Precision Logarithmic Number System,* IEEE Trans. on ASSP, February 1983.
16. E. E. Swartzlander et al., *Sign/Logarithmic Arithmetic for FFT Implementation,* IEEE Trans. on Computers, June 1983.
17. G. L. Sicuraza, *On Efficient Implementation of 2-D Digital Filters Using Logarithmic Number System,* IEEE ASSP, August 1983.
18. F. J. Taylor, *A Hybrid Floating Point Logarithmic Number System Processor,* IEEE Trans. on Circuits and Systems, January 1985.
19. F. J. Taylor, R. Gill, J. Joseph, and J. Radke, *A 20 Bit Logarithmic Number System Processor,* IEEE Trans. on Computers, February 1988.
20. E. P. Armendariz and S. McAdam, *Elementary Number Theory,* McMillan, 1980.
21. A. Peled and B. Liu, *A New Hardware Realization of Digital Filters,* IEEE Trans. on ASSP, ASSP-27, December 1974.
22. F. J. Taylor, *Digital Filter Design Handbook,* Marcel Dekker, Inc., NYC, 1984.
23. H. T. Kung and C. E. Leiserson, *Systolic Arrays (For VLSI),* Sparse Matrix Processings, SIAM, pp. 256–282, 1978.
24. F. J. Taylor et al., *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing,* IEEE Press, 1986.
25. F. J. Taylor and A. S. Rammarayanan, *An Efficient Residue-to-Decimal Converter,* IEEE Trans. on Circuits and Systems, December 1981.



Gin-Kou Ma (S'85) received the B.S. degree (1979) in Electrical Engineering from the University of Chinese Culture, Taiwan, Republic of China, and the M.S. (1984) and the Ph.D. (1989) degrees in Electrical Engineering from the University of Florida.

He was a member of the High-Speed Digital Architecture Laboratory at the University of Florida from 1985 through 1989. He joined the staff of Peregrine Engineering/The Athena Group Inc. in 1987 as a research scientist and he has been leading activities in digital signal processing and parallel processing system designs. He is a principal designer of MONARCH and SIGLAB software, where his responsibilities include algorithm development and DSP microprocessor interface. He is now a senior design engineer with the Athena Group Inc. His research interests are DSP algorithms and architectures, parallel computer architectures, and parallel algorithm designs.

Fred J. Taylor (M'69–SM'82) received the Ph.D. from the University of Colorado, Boulder, in 1969.

He is a Professor of Electrical Engineering and Computer and Information Science at the University of Florida, Gainesville, having been on the faculties of the University of Cincinnati, and University of Texas at El Paso. He has industrial experience with Texas Instrument, The Athena Group Inc, and Peregrine Engineering. He also is a consultant to industry and sits on several industrial advisory panels. He has authored over 100 papers, authored or co-authored five textbooks, authored two encyclopedia chapters, and and contributed to several other monographs, and holds four U.S. patents.