Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

# FAST MULTIPLICATION WITHOUT CARRY PROPAGATE ADDITION

Milos Ercegovac
Tomas Lang

September 1987
CSD-870047
(Rev.May 1988)

5/10/1988

# Fast Multiplication Without Carry-Propagate Addition

Miloš D. Ercegovac and Tomas Lang
Computer Science Department
University of California, Los Angeles

## Abstract

A common scheme for fast multiplication computes the accumulated partial products in redundant form (carry-save or signed-digit) and converts the result to conventional representation in the last step. This step requires a carry-propagate adder which is comparatively slow and occupies a significant area of the chip in a VLSI implementation. In this paper we report a multiplication scheme (LRCF — Left-to-Right, Carry-Free) that does not require this carry-propagate step. The LRCF scheme performs the multiplication most-significant bit first and produces a conventional sign-and-magnitude product by means of an on-the-fly conversion. The resulting implementation is fast and regular and is very well suited for VLSI. The LRCF scheme is presented for general radix $r$ and radix-4 implementations are described. Three different implementations are presented: one in which adders are of the signed-digit type, another in which carry-save adders are used, and a third that improves the speed by computing odd and even partial products concurrently.

## 1. Introduction

We describe a novel scheme for multiplication of two $n$-bit fractions producing an $n$-bit product (the most significant half of the full $2n$-bit product). A common scheme for fast multiplication computes the accumulated partial products in redundant form (carry-save or signed-digit) and converts the product to conventional representation in the last step [1] . This step requires a carry-propagate adder which is comparatively slow and occupies a significant area of the chip in a VLSI implementation [2]. In this paper we report a scheme that does not require this carry-propagate step. The basic characteristics of the proposed scheme are:

i) The recurrence uses the digits of the multiplier from most to least significant (left-to-right multiplication) [3]. The multiplier can be recoded into a suitable radix-$r$ representation to reduce the number of steps [4].

ii) The accumulated partial products are decomposed into two parts: the most significant part and the least significant part.

iii) To produce the product, the most significant portion of the accumulated partial products is converted to conventional form using a variation of the on-the-fly algorithm presented in [5], without the need of carry-propagate addition.

In the sequel, we refer to the proposed scheme as the LRCF (Left-to-Right, Carry-Free) multiplication. The LRCF scheme can be used both for sequential and combinational implementations. We concentrate here on the combinational case, since it provides the most in speed advantages. The resulting implementation is fast and regular and is very well suited for VLSI implementation.

We present the scheme for general radix $r$ and show implementations for radix 4. Three different implementations are presented: one in which adders are of the signed-digit type, another in which carry-save adders are used, and a third that improves the speed by computing odd and even partial products concurrently. We compare the LRCF scheme with conventional approaches in terms of implementation cost and delays. Error behavior and rounding of the LRCF scheme are discussed in general, with detailed considerations given in [6].

## 2. The LRCF multiplication algorithm

We consider multiplication of normalized fractions in the sign-and-magnitude representation. Let $X$ be the radix-2 representation of the normalized fractional magnitude $x$, such that

$$x = \sum_{i=1}^{n} X_i 2^{-i} \quad X_i \in \{0,1\} \tag{2.1}$$

and let $Y$ be the recoded radix-$r$ representation of the normalized fractional magnitude $y$, such that

$$y = \sum_{i=0}^{n/q} Y_i r^{-i} \quad Y_i \in \{-r/2,\ldots,r/2\} \quad (\textit{minimally redundant})$$

where, for simplicity, $r = 2^q$.

The LRCF multiplication algorithm is a recurrence that produces a sequence of two accumulated partial products ($w$ and $p$) as follows:

$$w[j] = r(\textit{fraction}(w[j-1] + xY_j)) \quad j = 0,\ldots,n/q \tag{2.2}$$

$$Z_j = \textit{integer}(w[j-1] + xY_j) \tag{2.3}$$

and

$$p[j] = p[j-1] + Z_j r^{-j} \tag{2.4}$$

The initial values are $w[-1] = p[-1] = 0$. Note that the algorithm uses the digits of the multiplier from most significant to least significant [3], unlike conventional multiplication schemes which use the digits from least significant to most significant.

To show that the LRCF algorithm performs multiplication observe that the sum of partial products after $k$ steps satisfies:

$$p[k] + w[k]\times r^{-k-1} = \sum_{i=0}^{k} xY_i \times r^{-i} \tag{2.5}$$

Consequently, after $n/q$ steps we obtain

$$p[n/q] + w[n/q]\times r^{-n/q-1} = \sum_{i=0}^{n/q} xY_i \times r^{-i} = xy \tag{2.6}$$

That is, $p[n/q]$ is the most significant part of the product while $w[n/q]$ is the least significant part.

A block diagram of one step of the recurrence is shown in Figure 1. A fast implementation requires the following:

i) Use of a *redundant adder* (either carry-save or signed-digit [7]) to produce $w[j]$. This results in a carry-free addition.

ii) Addition by *concatenation* to produce $p[j]$. That is,

$$p[j] = concat(p[j-1], P_j) \tag{2.7}$$

Since the maximum value of $Z_j$ in (2.3) is, in general, larger than $r-1$, to perform this concatenation it is necessary to recode $Z_j$ and $Z_{j+1}$ into $P_j$ in the range $[-(r-1), (r-1)]$. That is,

$$P_j = F(Z_j, Z_{j+1}) \quad P_j \varepsilon [-(r-1), (r-1)] \tag{2.8}$$

The details of the recoding depend on the range of $Z_j$, which is dependent on the type of redundant adder used to produce $w[j]$, as discussed in Sections 3 and 4.

iii) The *on-the-fly conversion* of the resulting signed-digit representation of $p[j]$ into a conventional representation $M[j]$. An algorithm to perform this conversion is given in [5]. In contrast with the traditional approach that performs the conversion by subtracting the negative part from the positive part (and therefore uses a carry-propagate adder), this on-the-fly scheme forms two numbers $a[j]$ and $b[j]$ such that, $a[j]$ is the converted number up to digit $j$ and $b[j]$ is $a[j]-r^{-j}$. When a new digit $P_j$ is produced, if it is positive or zero it is concatenated to $A[j-1]$ (digit-vector representing $a[j-1]$), while if it is negative $A[j]$ is obtained by concatenating $r-|P_j|$ to $B[j]$. That is,

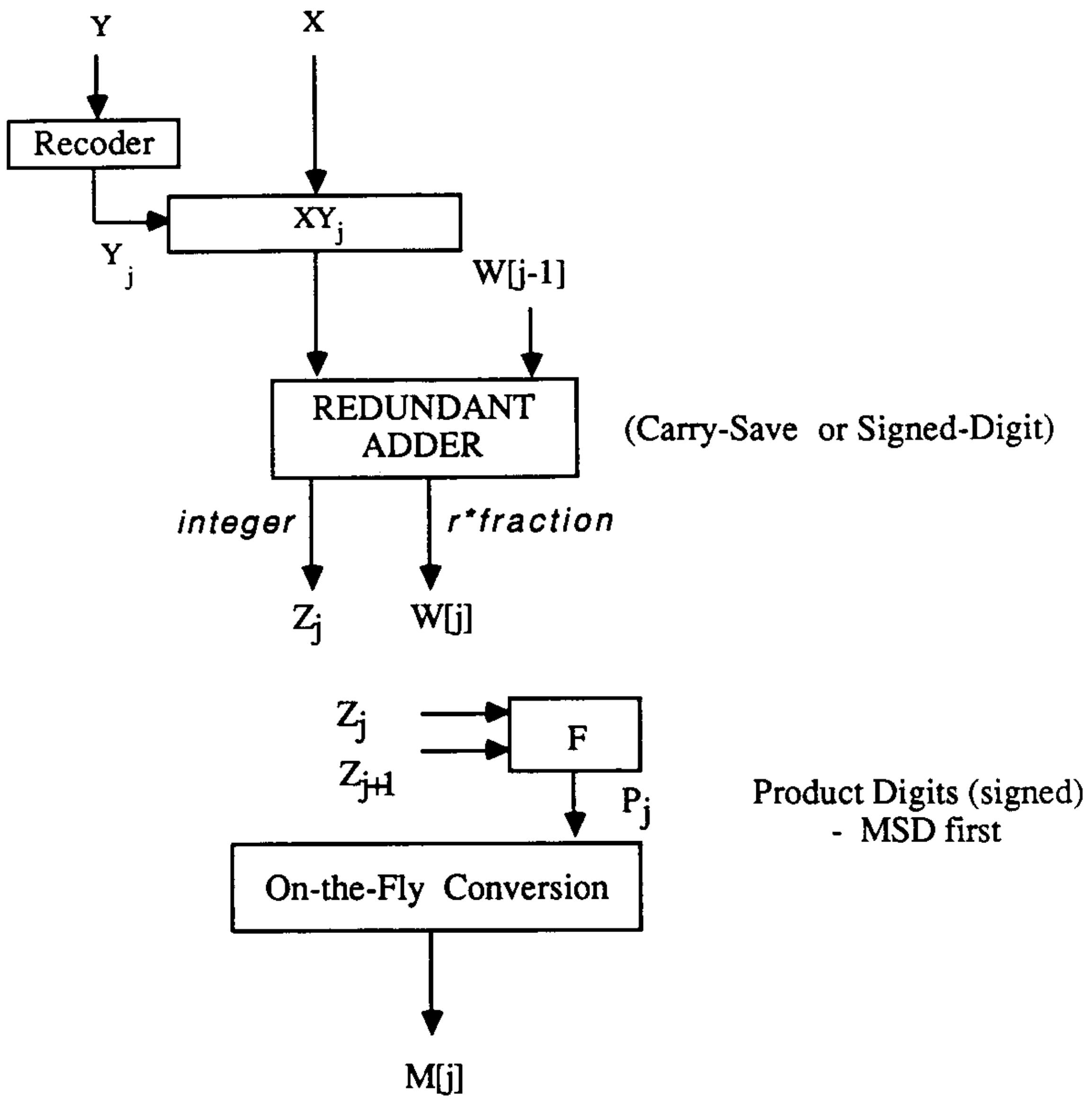


Figure 1. One Step of the LRCF Multiplication Scheme

$$A[j] = \begin{cases} concat(A[j-1], P_j) & \textbf{if } P_j \geq 0 \\ concat(B[j-1], (r - |P_j|)) & \textbf{if } P_j < 0 \end{cases} \tag{2.9}$$

$$B[j] = \begin{cases} concat(A[j-1], (P_j - 1)) & \textbf{if } P_j > 0 \\ concat(B[j-1], ((r-1) - |P_j|)) & \textbf{if } P_j \leq 0 \end{cases}$$

with the initial condition

$$A[-1] = B[-1] = 0 \tag{2.10}$$

After the last step, the most significant half of the product in conventional representation is $M[n/q] = A[n/q]$.

The step of Figure 1 can be used to implement either a sequential multiplier or a combinational one. The basic schemes for these two cases are illustrated in Figures 2a-2d, respectively, together with the corresponding conventional right-to-left approaches. The fundamental difference is shown in the timing diagram of Figure 2e: the two phases of multiplication - generation of partial products and formation of the final result in a conventional representation - are performed concurrently in the LRCF scheme and one after the other in conventional schemes.

The implementations of the LRCF scheme, shown in Figure 2, produce the most-significant half of the product without the need of a carry-propagate addition. In contrast, the right-to-left conventional schemes require a carry-propagate addition to obtain the most significant half. Consequently, the proposed scheme is faster in obtaining the most-significant part of the product, as required in many applications.

*Error and rounding schemes*

Since the result of the multiplication corresponds to the most-significant half of the product, an error is made with respect to the correct product. Several rounding schemes can be used to bound this error. The IEEE standard, for example, specifies four rounding schemes [8]. For instance, for a positive product the error for truncation is in the range $0 \leq \varepsilon_T < 2^{-n}$, while for rounding-to-nearest it is $-2^{-(n+1)} \leq \varepsilon_R \leq 2^{-(n+1)}$. These error ranges can be achieved in the conventional right-to-left schemes.

In the LRCF scheme, on the other hand, since the least-significant half of the product is left in redundant form, the error is somewhat larger than in the conventional right-to-left scheme. The actual range of the error depends on the type of redundant adder used, as discussed in detail later. For example, if a signed-digit adder is used the error is in the range $-2^{-n} < \varepsilon < 2^{-n}$. This doubling of the error might be acceptable in many applications. In cases in which a smaller error is required, the error can be reduced by increasing somewhat the number of digits of the calculated product and then performing a truncation or rounding of this result. In fact, the additional digits are part of the least-significant half that is in redundant form, so what is required is to as-

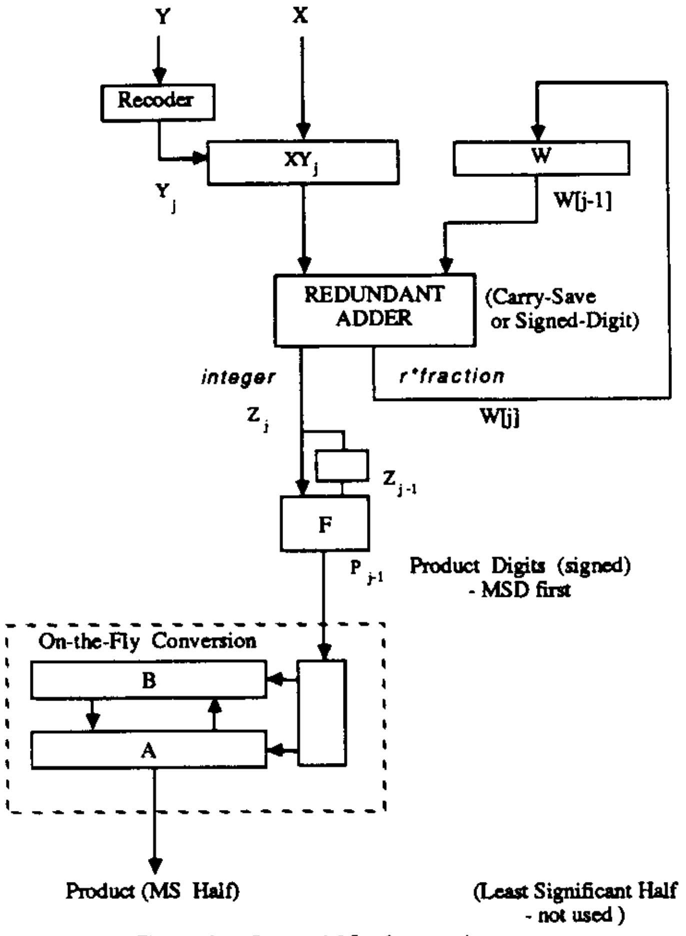


Figure 2a. Sequential Implementation of LRCF Algorithm

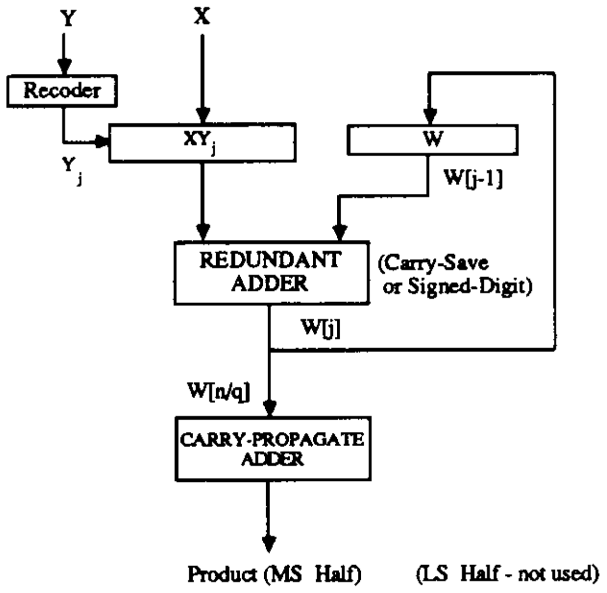


Figure 2b. Conventional Sequential Multiplier

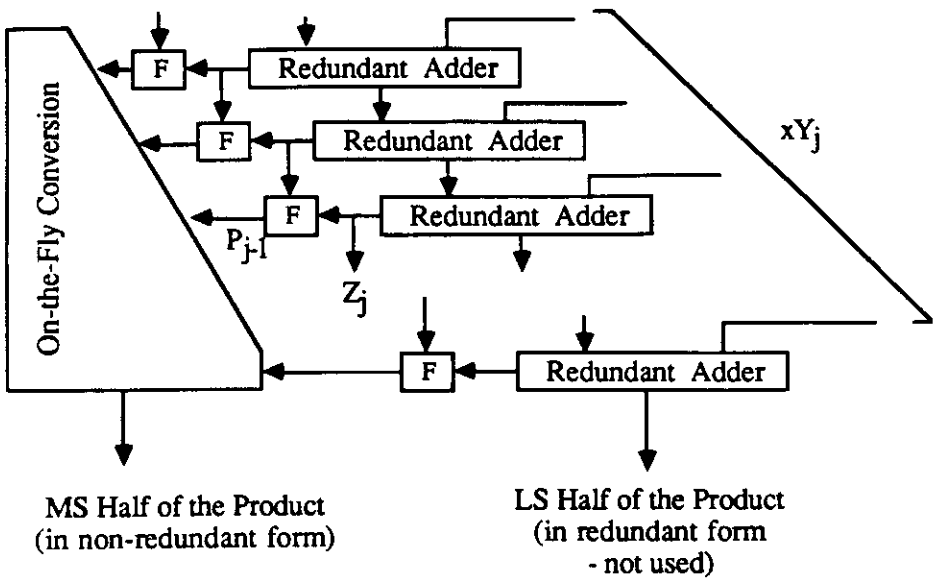

On-the-Fly Conversion
F
F
F
F
Redundant Adder
Redundant Adder
Redundant Adder
Redundant Adder
xY_j
P_j-1
Z_j
MS Half of the Product (in non-redundant form)
LS Half of the Product (in redundant form - not used)

Figure 2c. LRCF Combinational Multiplier

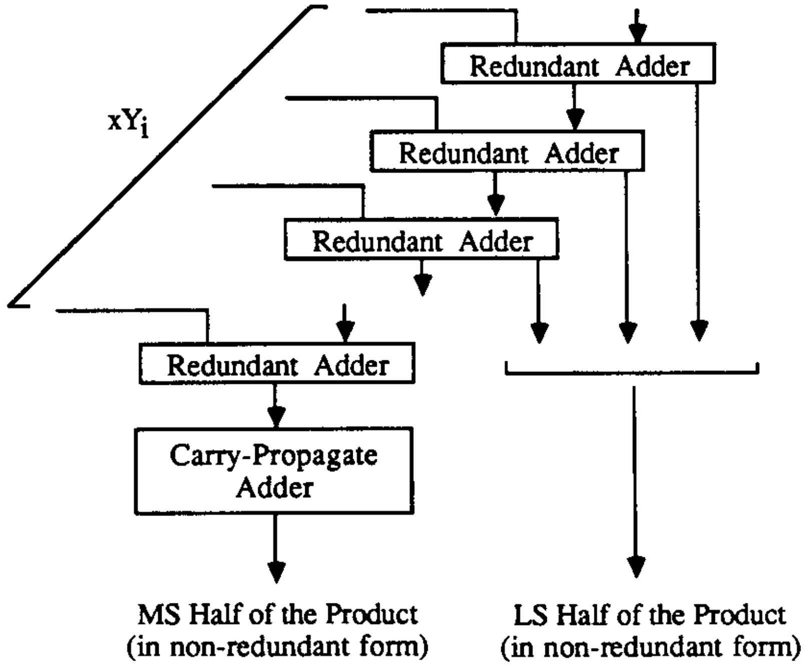

xY_i
Redundant Adder
Redundant Adder
Redundant Adder
Redundant Adder
Carry-Propagate Adder
MS Half of the Product (in non-redundant form)
LS Half of the Product (in non-redundant form)

Figure 2d. Conventional Combinational Multiplier

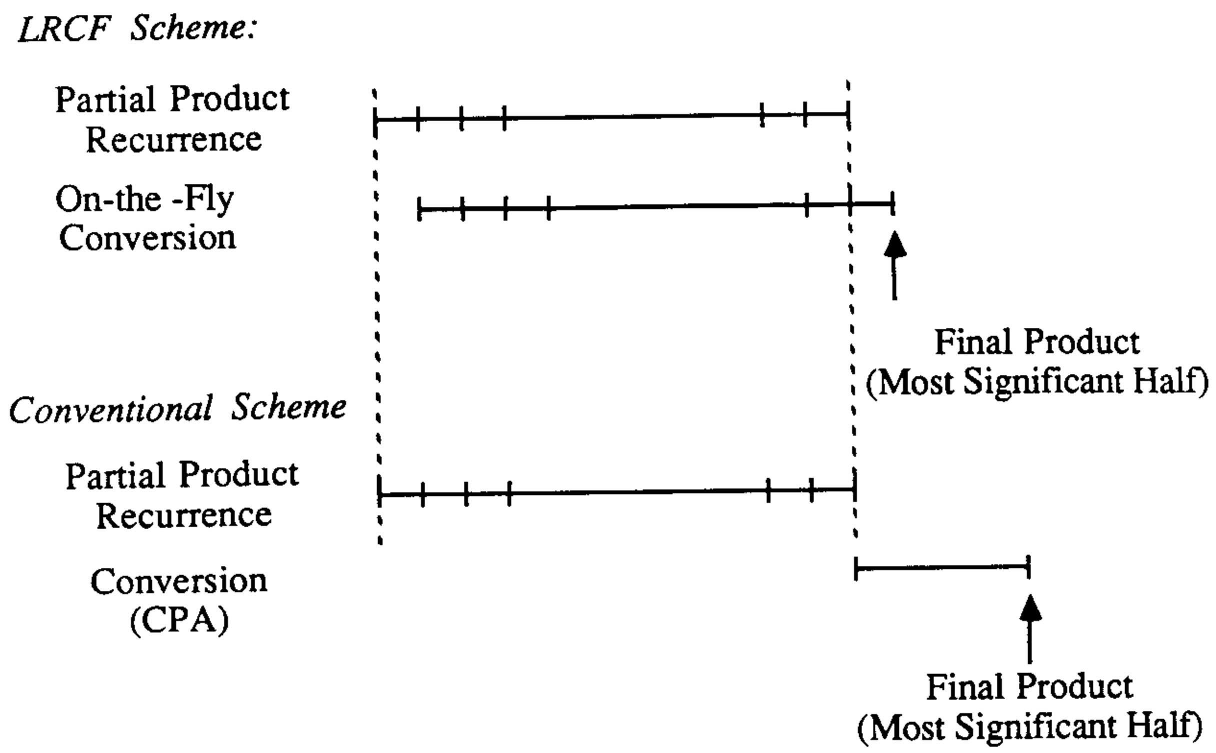

*LRCF Scheme:*
Partial Product Recurrence
On-the -Fly Conversion
Final Product (Most Significant Half)
*Conventional Scheme*
Partial Product Recurrence
Conversion (CPA)
Final Product (Most Significant Half)

Figure 2e. Timing Comparison Between LRCF and Conventional Multiplication Schemes

similate these few digits with a short carry-propagate adder.

Other rounding schemes, with different error characteristics, can also be obtained. As an example of one of these schemes, in the Cray X-MP supercomputer a conventional right-to-left multiplication of 48-bit operands is performed (and a 48-bit result is obtained)[9]. To reduce the size of the carry-save adder array required, the array is limited to a width of 56 bits (instead of 96 bits). To reduce the average truncation error and compensate for the bits not computed, a constant equal to $9 \times 2^{-56}$ is added. Of course, in the Cray case, since the multiplication is right-to-left a carry-propagate addition of 48 bits is required to obtain the result (most-significant half). This addition is not required in the LRCF scheme.

If, on the other hand, it is desired to obtain the error ranges required by the IEEE standard, two options exist:

a) Increasing by one the number of bits of the representation. That is, the left-to-right scheme with $n+1$ bits has the same error range as the right-to-left scheme with $n$ bits. This would be a good solution for some special-purpose processors.

b) To reduce the error so that it is possible to use the standard rounding schemes with $n$ bits. This can be done by obtaining additional information from the least-significant portion of the product. As an example consider the following product obtained from the left-to-right multiplier using signed-digit adders:

$$p\,(most\ \ significant) = 0.1001$$

$$p\,(least\ \ significant) = 000\bar{1} \quad (\bar{1} = -1)$$

In this case, correct truncation cannot be performed by just knowing the most significant part of the product, but it is necessary to know also the sign of the least-significant portion. One way of obtaining this sign would be to convert this portion to conventional representation. However, a carry-propagate addition would be required for this, which eliminates the main advantage of the scheme. A better solution is to perform a partial right-to-left multiplication concurrently with the main left-to-right one, as illustrated in Figure 3. Of course, this increases the hardware required but it does not increase the time of multiplication. The details of this scheme depend on the type of redundant adder used, and are discussed in the following sections and in [6].

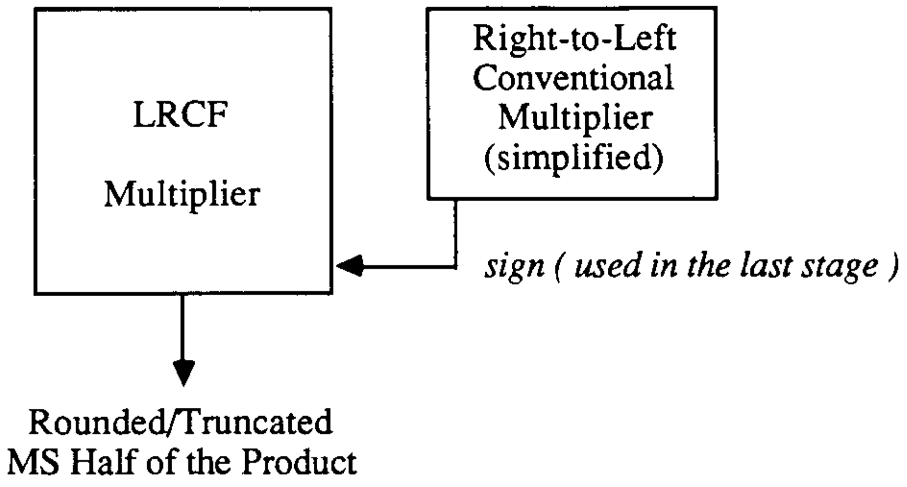


Figure 3. A Scheme for Correct Round-Off Process

*Example 2.1*

We now give an example of the execution of LRCF multiplication. The additions are performed in radix-4 signed-digit.

$x = .11010110_2 = (.3112_4)$ $y = .11010111_2$ Recoded $Y = 1.\bar{1}12\bar{1}_4$

$w[0] = xY_0$ $Z_0 = 0$ ( since $|x| < 1$ and $Y_0 \in \{0,1\}$) $A[-1] = B[-1] = 0$

| $j$ | $w[j-1]$<br>$xY_j \times 4^{-j}$ | $Z_j$ | $P_{j-1}$ | $A[j-1]$<br>$B[j-1]$ |
|---|---|---|---|---|
| 1 | $3\,1\,1\,2$<br>$=\bar{3}\,\bar{1}\,\bar{1}\,\bar{2}$ | 3 | 1 | 1<br>0 |
| 2 | $=\bar{2}\,0\,1\,\bar{2}$<br>$==3\,1\,1\,2$ | -2 | -1 | 0.3<br>0.2 |
| 3 | $==3\,2\,\bar{1}\,2$<br>$==1\,2\,2\,3\,0$ | 5 | -1 | 0.23<br>0.22 |
| 4 | $===0\,2\,1\,0$<br>$====\bar{3}\,\bar{1}\,\bar{1}\,\bar{2}$ | 0 | 1 | 0.231<br>0.230 |
| 5 | $====\bar{1}\,0\,\bar{1}\,\bar{2}$ | -1 | 0 | 0.2310 |

The final result is $M[4] = A[4] = 0.2310$. The error is $\varepsilon = w[4] \times 4^{-5} = -0.00001012$. The rules for obtaining $P_j = F(Z_j, Z_{j+1})$ are discussed in Sections 3 and 4.

□

## 3. Radix-4 Implementation with signed-digit addition

We now present the combinational implementation of the LRCF scheme for a radix-4 multiplication unit using a linear array of signed-digit adders [7] for the computation of $w$. For radix-4 the recurrences are

$$w[j] = 4(\textit{fraction}(w[j-1] + xY_j)), \quad j = 0,\ldots,n/2, \quad w[-1] = 0, \quad Y \in \{-2,-1,0,1,2\} \tag{3.1}$$

$$Z_j = \textit{integer}(w[j-1] + xY_j) \tag{3.2}$$

and

$$p[j] = p[j-1] + Z_j 4^{-j} = \textit{concat}(p[j-1], P_j), \quad p[-1] = 0 \tag{3.3}$$

where $P_j$ is obtained from $Z_j$ and $Z_{j+1}$ by signed-digit addition [7] as described next.

To determine the range of $Z_j$, observe that

$$Z_j = integer(w[j-1] + xY_j) = integer(w[j-1]) + integer(xY_j) + t_0 \tag{3.4}$$

where $t_0 \in \{-1,0,1\}$ is a transfer digit resulting from the signed-digit addition of the fractions of $w[j-1]$ and $xY_j$. Moreover, the most significant digit of the fraction of the signed-digit sum

$$w[j-2] + xY_{j-1} \tag{3.5}$$

is in the range $\{-3,...,3\}$. Therefore,

$$integer(w[j-1]) = integer(4 \times fraction(w[j-2] + xY_{j-1})) \tag{3.6}$$

is in the range $\{-3,...,3\}$. Since $|x| < 1$ is a fraction in conventional form and the recoded multiplier digits $Y_j$'s are in the range $\{-2,...,2\}$, $integer(xY_j)$ is in the range $\{-1,0,1\}$. Consequently, the range of $Z_j$ is $[-5,5]$ as illustrated in Figure 4.

The implementation consists of three parts as follows:

1) A linear array of signed-digit adders to compute the $w's$. This array includes the recoder of $Y$ and the selection of the multiples of $x$ (Figure 5a).

2) A linear array of modules TS to compute the partial products $p[j]$'s (Figure 5b) from left to right. In the partial product computation (3.3), a direct addition of $Z_j$'s would require carry propagation. To perform these additions without carry propagation, $Z_j$ (with range $[-5,5]$) is recoded into $t_{j-1}$ and $s_j$ such that

$$Z_j = 4t_{j-1} + s_j \tag{3.7}$$

with

$$|t_{j-1}| \le 1 \quad \text{and} \quad |s_j| \le 2 \tag{3.8}$$

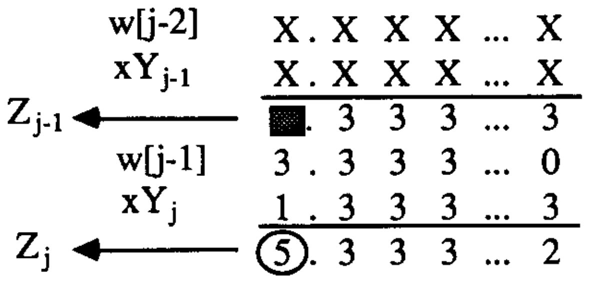
A possible recoding of $Z_j$ is

| $Z_j$ | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $t_{j-1}$ | -1 | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| $s_j$ | -1 | 0 | 1 | -2 | -1 | 0 | 1 | 2 | -1 | 0 | 1 |

Then $P_j = t_j + s_j$ is the $j$th product digit. Because of the ranges (3.8), $P_j \in \{-3,...,3\}$. Since

$$p[j] = \sum_{i=0}^{j} P_i 4^{-i} \quad P_i \in \{-3,...,3\} \tag{3.9}$$

the partial products $p[j]$'s are computed by concatenation



(similarly for negative values)

Figure 4. Range of Z Digits

$$p[j] = concat(p[j-1], P_j) \tag{3.10}$$

thus avoiding carry propagation.

3) The signed-digit representation of the product $p[n/2]$ is converted to conventional representation $M[n/2]$ using a combinational variation of the on-the-fly algorithm [5,8], as shown in Figure 5b. Instead of using the two conditional forms $A$ and $B$, described in Section 2, we keep only $A$ together with control signals $D_k[j]$ associated with each $A_k$ (digit of $A$), to determine whether the final digit $M_i$ is $A_i$ or $(A_i - 1)$ *mod* 4. The meaning of the control signals is given in Table 3.1.

Table 3.1: Conversion Control Signals

| $D_k[j]$ | Decision at level $j$ about value of product digit $M_k[n/2]$ |
|---|---|
| $u$ | undecided ($M_k[n/2] = A_k$) |
| $n$ | decided: no change ($M_k[n/2] = A_k$) |
| $d$ | decided: decrement ($M_k[n/2] = (A_k - 1)\ mod\ 4$) |

A high-level description of the conversion process is

$$A_j = P_j mod\ 4 = \begin{cases} P_j & \textbf{if } P_j \geq 0 \\ 4 + P_j & \textbf{if } P_j < 0 \end{cases} \tag{3.11}$$

Initially,

$$D_j[j] = u \tag{3.12}$$

For $k < j$,

$$D_k[j+1] = \begin{cases} u & \textbf{if } D_k[j] = u \textbf{ and } P_j = 0 \\ n & \textbf{if } D_k[j] = n \textbf{ or } (D_k[j] = u \textbf{ and } P_j > 0) \\ d & \textbf{if } D_k[j] = d \textbf{ or } (D_k[j] = u \textbf{ and } P_j < 0) \end{cases} \tag{3.13}$$

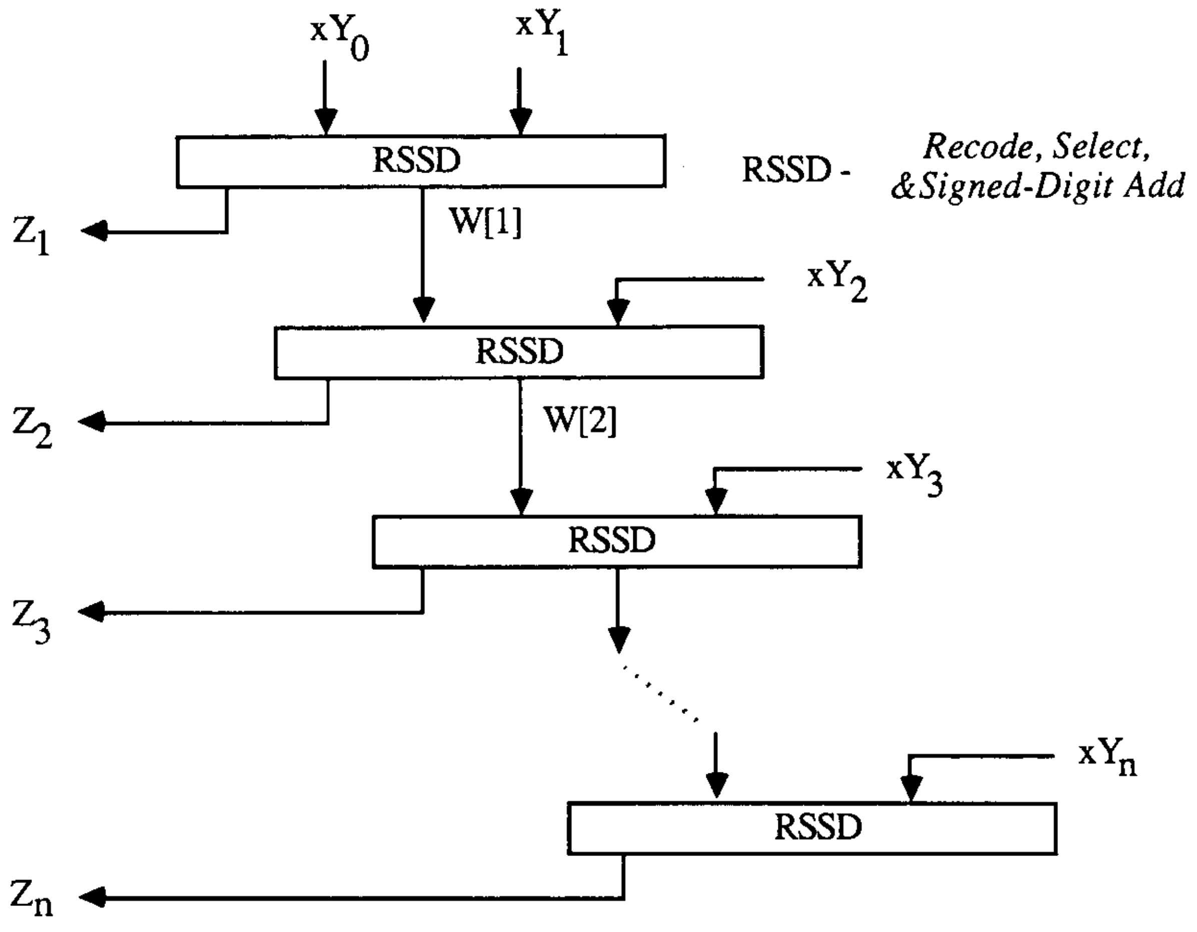
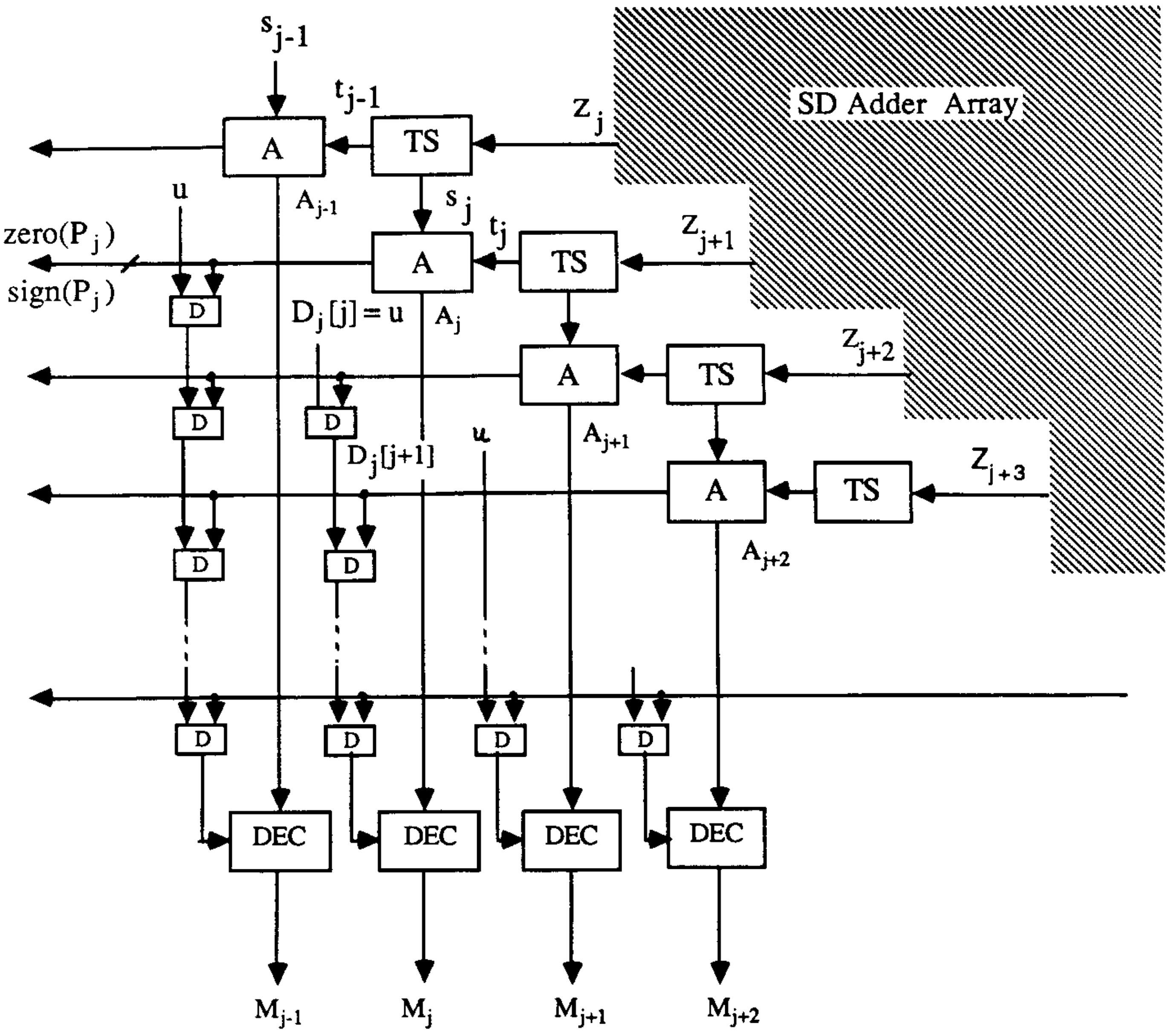


Figure 5a. Linear Array of Signed-Digit Adders for Computing W's and Z's



Final Product (Most Significant Half)

Figure 5b. Implementation of Result Generation
- A Segment

Consequently, $D_k[j+1]$ depends on $D_k[j]$ and on the signals

$$zero(P_j) = \begin{cases} 1 & \textbf{if } P_j = 0 \\ 0 & \textbf{otherwise} \end{cases} \qquad sign(P_j) = \begin{cases} 1 & \textbf{if } P_j < 0 \\ 0 & \textbf{otherwise} \end{cases} \tag{3.14}$$

The on-the-fly conversion of $P = 2003002\overline{1}$ into $M = 20023313$ is illustrated in Table 3.2.

Table 3.2 Example of On-the-Fly Conversion

| $j$ | $D_1[j]$ $P_1$ $A_1$ $M_1$ | $D_2[j]$ $P_2$ $A_2$ $M_2$ | $D_3[j]$ $P_3$ $A_3$ $M_3$ | $D_4[j]$ $P_4$ $A_4$ $M_4$ | $D_5[j]$ $P_5$ $A_5$ $M_5$ | $D_6[j]$ $P_6$ $A_6$ $M_6$ | $D_7[j]$ $P_7$ $A_7$ $M_7$ | $D_8[j]$ $P_8$ $A_8$ $M_8$ |
|---|---|---|---|---|---|---|---|---|
| | 2 | | | | | | | |
| 1 | $u$ 2 | 0 | | | | | | |
| 2 | $u$ | $u$ 0 | 0 | | | | | |
| 3 | $u$ | $u$ | $u$ 0 | 3 | | | | |
| 4 | $n$ | $n$ | $n$ | $u$ 3 | 0 | | | |
| 5 | $n$ | $n$ | $n$ | $u$ | $u$ 0 | 0 | | |
| 6 | $n$ | $n$ | $n$ | $u$ | $u$ | $u$ 0 | $\overline{2}$ | |
| 7 | $n$ | $n$ | $n$ | $d$ | $d$ | $d$ | $u$ 2 | $\overline{1}$ |
| 8 | $n$ | $n$ | $n$ | $d$ | $d$ | $d$ | $d$ | $u$ 3 |
| | 2 | 0 | 0 | 2 | 3 | 3 | 1 | 3 |

Consequently, the conversion part is composed of

i) The modules A that generate $A_j$ according to (3.11), and $zero(P_j)$ and $sign(P_j)$ signals.

ii) The modules D that update $D_k[j]$ according to (3.13).

iii) The modules DEC that decrement $A_k$ (modulo 4) if $D_k[n/2] = d$.

*Bit-level implementation and comparison with conventional schemes*

The modules and their connections are indicated in Figures 5a and 5b. Their bit-level implementation is discussed in Appendix A. The LRCF and a conventional scheme are similar regarding the following parts: the binary-to-radix-4 multiplier recoder, the selection of multiples of the multiplicand (±2×, ±1×, 0×), and the array of redundant adders for the accumulation of partial products. These adders are composed of signed-digit adder modules [12,13,14] instead of full-adders. According to [12], the signed-digit adders are similar in area and delay to conventional carry-save adders. The principal difference is in producing the final product in conventional representation from a sum of partial products in redundant form: the LRCF scheme uses an on-the-fly converter while in a conventional scheme a CPA adder is required.

Since a product of $n$ bits is to be computed, those digits of the array that do not influence the result can be eliminated. As shown in Figure 6, for a radix-4 recoding of the multiplier, the first half of the array is of full precision and from then on, the number of radix-4 adders decreases by one per level. A similar reduction in size of the adder array can be achieved in a conventional right-to-left multiplier, as done for example in the Cray X-MP processor [9], in which case the error in multiplication is similar to that produced by the LRCF scheme.

*Delay of the scheme and comparison with conventional schemes*

The delay of the generation of the product is composed of the following:

i) Recoding and forming the multiples of the multiplicand.

ii) Delay to obtain the last partial product $w[n/2]$ in the signed-digit adder array. This corresponds to $(n/2)-1$ signed-digit adders.

iii) Delay to produce the last $zero(P_j)$ and $sign(P_j)$ signals.

iv) Delay to determine the value of the last $D$'s.

v) Delay of digit decrementing.

In comparison, in a conventional right-to-left multiplier the delay corresponds to i) and ii) above plus a carry-propagate addition. Consequently, the scheme presented here is faster by the difference in delay between the carry-propagate adder and the sum of the delays iii) to v) above. Since the CPA delay is at best $O(\log_2 n)$ logic levels and steps iii) to v) can be implemented in a couple of levels, this difference is significant, especially for larger operand precision. To reduce the total delay, the last step of the LRCF scheme can be optimized for speed.

*Example 3.1*

To illustrate the implementation we show an example of an 8x8 bits multiplication. The additions are performed in radix-4 signed-digit.

$x = .11010110_2 = .3112_4$ $y = .11010111_2 = .3113_4$ Recoded Y = $1.\bar{1}12\bar{1}_4$ $w[0] = xY_0$ $Z_0 = 0$ $(s_0 = 0)$

| $j$ | $w[j-1]$ $xY_j4^{-j}$ | $Z_j$ | $t_{j-1}$ | $s_j$ | $P_{j-1}$ | $A[j-1]$ | $D[j-1]$ |
|---|---|---|---|---|---|---|---|
| 1 | $3\,1\,1\,2$ $=\bar{3}\,\bar{1}\,\bar{1}\,\bar{2}$ | 3 | 1 | -1 | 1 | 1. | *u* |
| 2 | $\bar{2}\,0\,1\,\bar{2}$ $==3\,1\,1\,2$ | -2 | 0 | -2 | -1 | 1.3 | *d u* |
| 3 | $==3\,2\,\bar{1}\,2$ $==1\,2\,2\,3\,0$ | 5 | 1 | 1 | -1 | 1.33 | *d d u* |
| 4 | $===0\,2\,1\,0$ $====\bar{3}\,\bar{1}\,\bar{1}\,\bar{2}$ | 0 | 0 | 0 | 1 | 1.331 | *d d n u* |
| 5 | $====\bar{1}\,0\,\bar{1}\,\bar{2}$ | -1 | 0 | -1 | 0 | 1.3310 | *d d n u u* |

*Decrement digits of A[4] according to D[4]:* $M[4] = 0.2310$

*Check:* $.3112 \times .3113 = .2310 + .0000\bar{1}0\overline{12}$ (radix-4) □

*Error and rounding schemes*

As mentioned in Section 2, for a result of $n$ fractional bits, the error in the LRCF scheme is

$$-2^{-n} < Error < 2^{-n} \tag{3.15}$$

This error is larger than what is allowed by the rounding schemes in the IEEE Standard [8]. However, it is suitable for many applications.

To perform the IEEE rounding it is necessary to determine the sign of the signed-digit remainder (this corresponds to the least-significant part of the product) and to identify the case when this remainder has value 0 (for getting unbiased rounding-to-nearest). Two possibilities exist for obtaining these values, namely

i) Use a sign-detection and zero-detection circuit. The corresponding network is simpler than a carry-propagate adder, but corresponds roughly to the computation of a carry-out signal. A possible fast implementation consists of a network of $\log_k n$ levels, where $k$ is the maximum fan-in of the gates. The delay of this network can be avoided by using the next option.

ii) The sign can be computed concurrently with the left-to-right multiplication by a simplified right-to-left multiplication. The details of the implementation, as well as the use of the obtained values in the alternative rounding schemes, are given in [6].

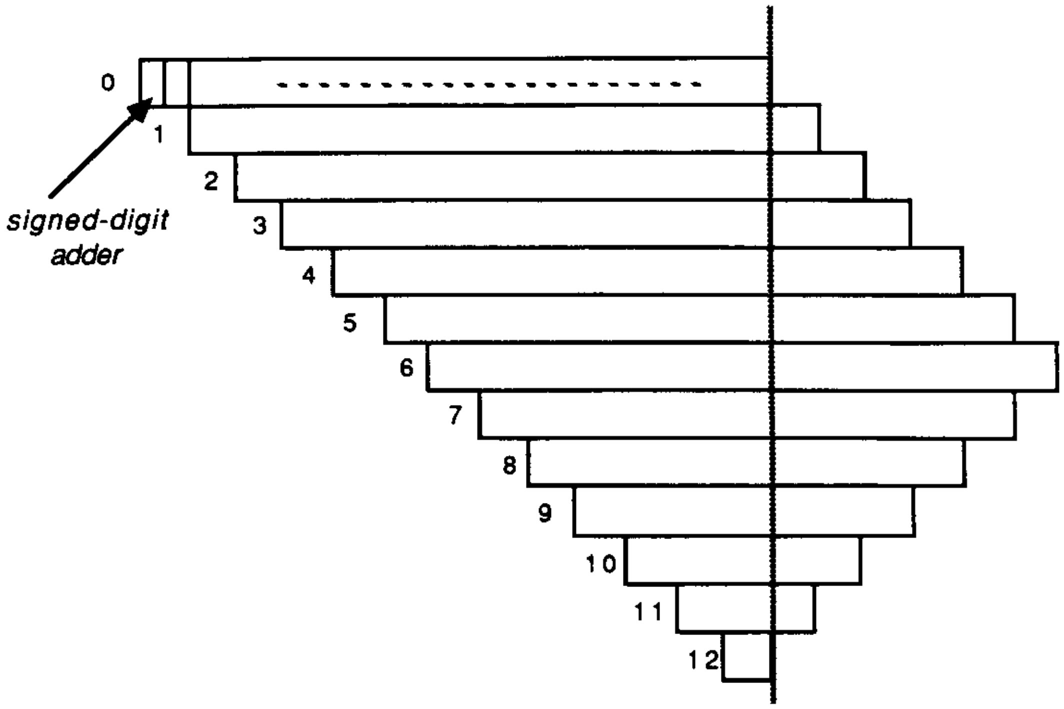


Figure 6. Reduced Width of Signed-Digit Adder Array

## 4. Radix-4 Implementation with Carry-Save Adders in the Multiplication Array

A second possible implementation of the LRCS multiplier uses a linear array of carry-save adders for the computation of the $w's$. The advantage of this implementation with respect to that presented in the previous section is that a carry-save adder is composed of full adders, which might result in a simpler implementation than using signed-digit adders. We now describe the computation of $s$ and $t$ for this case.

In this case $xY_j$ is represented in the 2's complement system, so that the corresponding sign extension when $Y_j$ is negative has an influence on the already computed digits $Z_i$. To avoid the recomputation of these digits, instead of extending the sign, when $xY_j$ is negative a value 2 is subtracted from the present digit (Figure 7a). The resulting expression for $Z_j$ is

$$Z_j = integer(w[j-1] + q) - 2 \times sign(Y_j) = u_j - 2 \times sign(Y_j) \tag{4.1}$$

where $q$ is $xY_j$ without the sign extension.

The range of $Z_j$ is now from +8 to -2 (Figure 7b). This range is suitable for the use of signed-digit concatenation for $p[j]$. However, the implementation can be simplified by scaling the multiplicand into the range $|x| \le 1/2$ (this is achieved by dividing the multiplicand by 2 and adding one more recurrence step). In this case (Figure 7c) the range of $Z_j$ is from 7 to -1 and the expression to compute it is

$$Z_j = u_j - sign(Y_j) \tag{4.2}$$

where

$$u_j = 2(PS_{-1}[j-1] + PC_{-1}[j-1]) + PS_0[j-1] + PC_0[j-1] + PC_0[j] \tag{4.3}$$

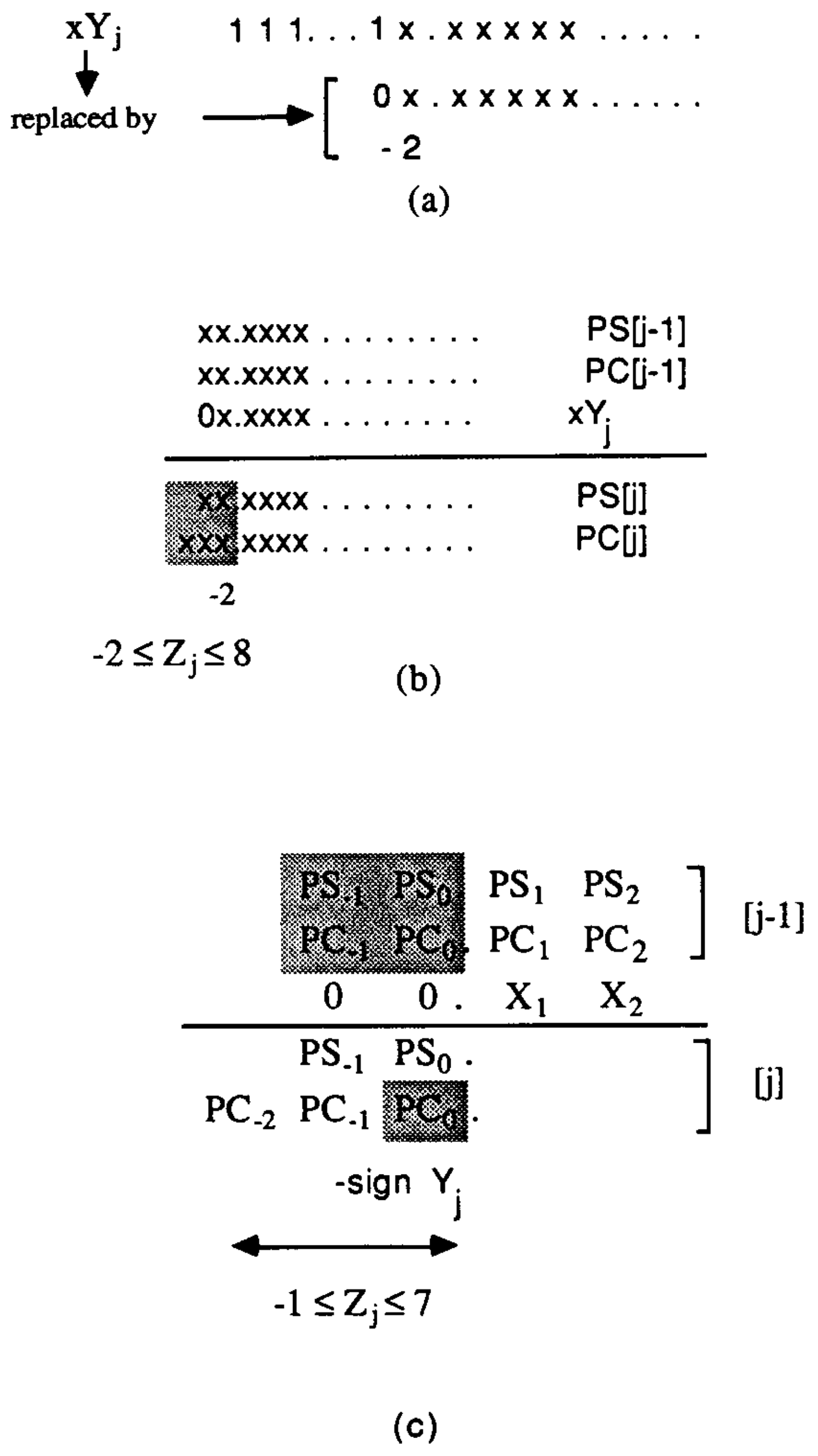
assuming that the carry-save form of $w[j]$ is $(PC[j], PS[j])$.

As discussed in the previous section, to perform the computation of $p[j+1]$ using signed-digit concatenation, we recode $Z[j]$ into $t_{j-1}$ and $s_j$ so that

$$Z_j = 4t_{j-1} + s_j \tag{4.4}$$

and $|t_j + s_j| \le 3$

We now present a possible recoding. As shown in Figure 8, the TSA block computes $t_{j-1}$ and $s_j$ and has as inputs $PS_{-1}[j-1], PC_{-1}[j-1], PS_0[j-1], PC_0[j-1], PC_0[j], sign(Y_j)$. The specific recoding function is selected to simplify the implementation. To make $t_{j-1}$ dependent only on $u_j$ (and not on $sign(Y_j)$), the recoding is done in two steps as follows:



xY_j
1 1 1. . . 1 x . x x x x x . . . . .
replaced by
0 x . x x x x x . . . . . .
- 2
(a)
xx.xxxx . . . . . . . . PS[j-1]
xx.xxxx . . . . . . . . PC[j-1]
0x.xxxx . . . . . . . . xY_j
xx.xxxx . . . . . . . . PS[j]
xxx.xxxx . . . . . . . . PC[j]
-2
-2 ≤ Z_j ≤ 8
(b)
PS_-1 PS_0 PS_1 PS_2
PC_-1 PC_0 PC_1 PC_2
[j-1]
0 0 . X_1 X_2
PS_-1 PS_0 .
PC_-2 PC_-1 PC_0 .
[j]
-sign Y_j
-1 ≤ Z_j ≤ 7
(c)

Figure 7. $Z_j$ ranges

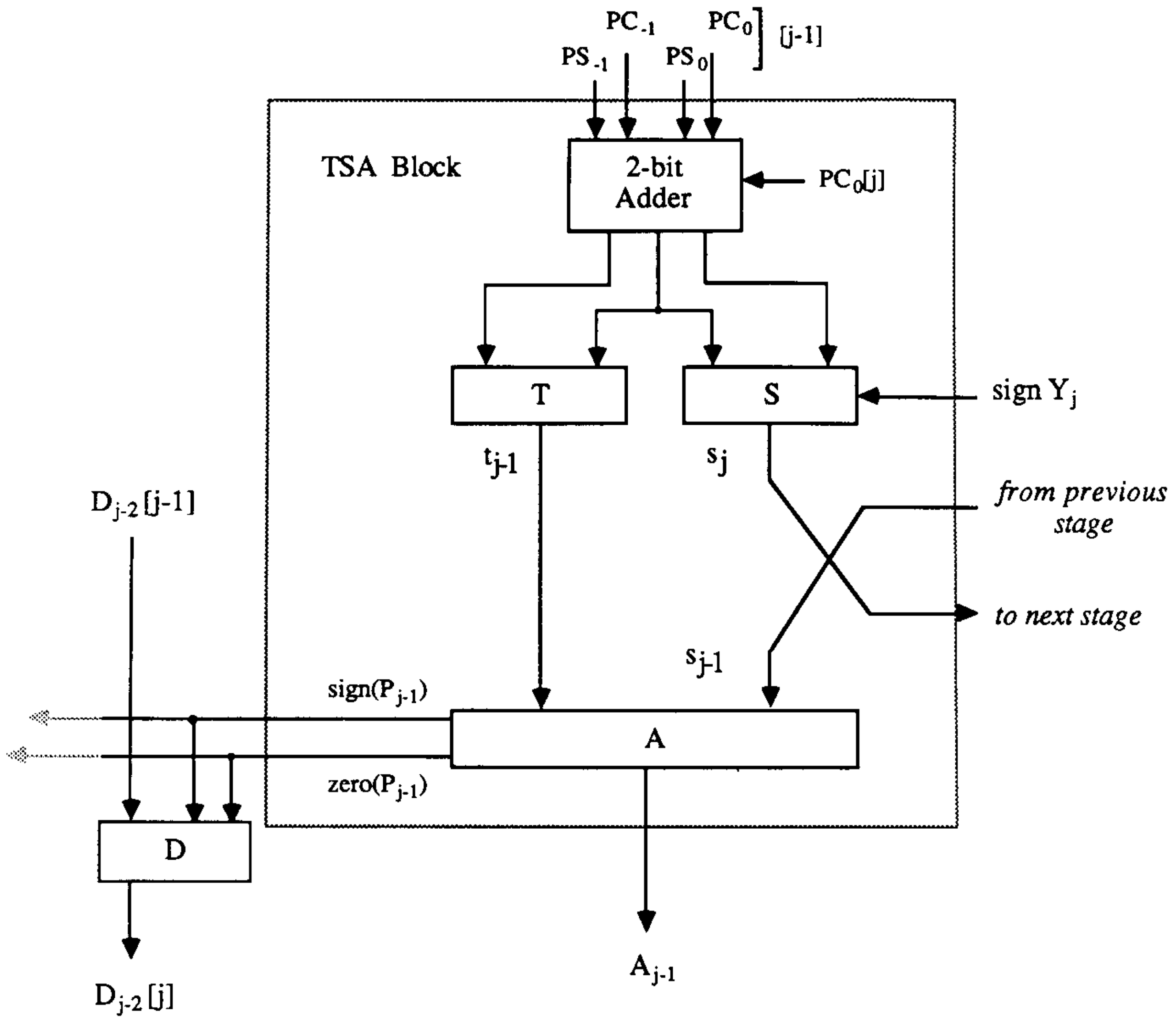


Figure 8. Bit-Level Implementation of TSA Block

i) Calculate $t_{j-1}$ and a temporary $s_j^t$ such that

$$u_j = 4t_{j-1} + s_j^t \tag{4.5}$$

The corresponding table is

| $u_j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $t_{j-1}$ | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
| $s_j^t$ | 0 | 1 | -2 | -1 | 0 | 1 | -2 | -1 |

ii) Subtract $sign(Y_j)$, so that

$$s_j = s_j^t - sign(Y_j) \tag{4.6}$$

which produces a $s_j$ in the range -3 to 1. Since the range of $t_j$ is 0 to 2, the property of carry-free addition is satisfied.

The values $t_j$ and $s_j$ are then used to compute $A_j$ and $sign(P_j)$ and $zero(P_j)$, which are used in the conversion. This is done as in the signed-digit case presented in Section 3.

*Bit-level implementation and comparison*

The implementation of the TSA block is divided into several modules as shown in Figure 8. In addition, modules A and D perform on-the-fly conversion. The bit-level implementation of these modules is discussed in Appendix B.

In comparison with the signed-digit implementation, this carry-save case is advantegeous because the carry-save adders (full adders) are simpler than the signed-digit adders.

*Delay of the scheme and comparison*

The delay of producing the most significant half of the product has the same components as in the signed-digit implementation (Section 3) except that 3-to-2 carry-save adders are used in the array instead of the signed-digit adders. Again, the difference in delay between the LRCF multiplier and a conventional right-to-left multiplier is equal to the delay of the last stage and the delay of an $n$-bit carry-propagate adder. As mentioned in Section 3, to reduce the total delay of the LRCF scheme, the last step can be optimized for speed.

*Example 4.1*

To illustrate the implementation we show an example of an 8x8 bits multiplication.

$x = .10101011_2$   $y = .11001001_2$  Recoded $Y = 1.\bar{1}1\bar{2}1_4$  Shifted $X = .010101011_2$

$PS[0] = xY_0$ $PC[0] = 0$ $t_{-1} = 0$ $s_0 = 0$

| $j$ | $PS[j-1]$<br>$PC[j-1]$<br>$xY_j$ | $u_j$ | $sign(Y_j)$ | $t_{j-1}$ | $s_j$ | $P_{j-1}$ | $A[j-1]$ | $D[j-1]$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 010101011<br>000000000<br>==101010101 | 1 | 1 | 0 | 0 | 0 | 0 | *u* |
| 2 | ==111111001<br>==00000100<br>====01010101 | 3 | 0 | 1 | -1 | 1 | 0.1 | *n u* |
| 3 | ====10110111<br>====10100000<br>======010101 | 5 | 1 | 1 | 0 | 0 | 0.10 | *n u u* |
| 4 | ======00001<br>======10101<br>========010 | 2 | 0 | 1 | -2 | 1 | 0.101 | *n n n u* |
| 5 | ========11<br>========01 | 4 | 0 | 1 | 0 | -1 | 0.1013 | *n n n d u* |

*Decrement* $A[4]$ and attach last digit: $0.1013_4 = 00.01\ 00\ 00\ 11\ 00_2$

*Shift:* 00.10 00 01 10

*Check:* .10101011×.11001001=.1000011001000011

□

*Error and Rounding Schemes*

In this case, since the least significant part of the product is left in carry-save form, the error is always positive and its magnitude is

$$0 \leq Error \leq 2^{-n+1} \tag{4.7}$$

Similar to the signed-digit case, this error is larger than what is required for the IEEE rounding schemes. However, as discussed before, it might be acceptable for many applications.

To be able to perform the IEEE rounding schemes, it is necessary to detect the case when the value of the redundant least-significant part is larger or equal to $2^{-n}$ and when it is equal to zero. Similar to the signed-digit case, two options exist

i) To have a carry-detection and zero detection network. However, this network adds significantly to the delay.

ii) To generate the carry and the zero signal by a simplified right-to-left multiplication that is performed concurrently with the main LRCF multiplication. The details of the implementation of this approach, as well as the use of the values obtained in the different rounding schemes, are described in [6].

## 5. Radix-16 LRCF multiplier formed by even-odd radix-4 arrays

One way of increasing the speed of a multiplication array is to increase the radix. The even-odd scheme effectively produces the speed of a radix-16 array while maintaining the advantages of radix-4 arrays of requiring only multiples ±2, ±1 and 0. This type of array has been used for the right-to-left approach with carry-propagate adder [11]. Here we extend its use to the left-to-right scheme without carry-propagate adder. We present the scheme for the case in which the array uses signed-digit adders; it can be readily transformed to the carry-save case.

The even-odd scheme divides the array into two subarrays, as shown in Figure 9. The even subarray performs the addition of the $xY_j$ with $j$ even, while the odd subarray adds those with $j$ odd. Let us call $w[j]$ the partial product for the even subarray and $v[j]$ that for the odd one. The corresponding LRCF multiplication algorithm becomes

$$w[j+2] = 16(\mathit{fraction}(w[j] + xY_j) \quad j \ \ \mathit{even} \tag{5.1}$$

$$v[i+2] = 16(\mathit{fraction}(v[i] + 4xY_i) \quad i \ \ \mathit{odd}$$

Note the multiplication by 4 in the odd recurrence, which is needed because of the different weight of the corresponding $xY_j$. The corresponding $Z's$ are computed as

$$Z_j(\mathit{even}) = \mathit{integer}(w[j] + xY_j) \tag{5.2}$$

$$Z_i(\mathit{odd}) = \mathit{integer}(v[i] + 4xY_i)$$

From these expressions we conclude that the corresponding ranges are

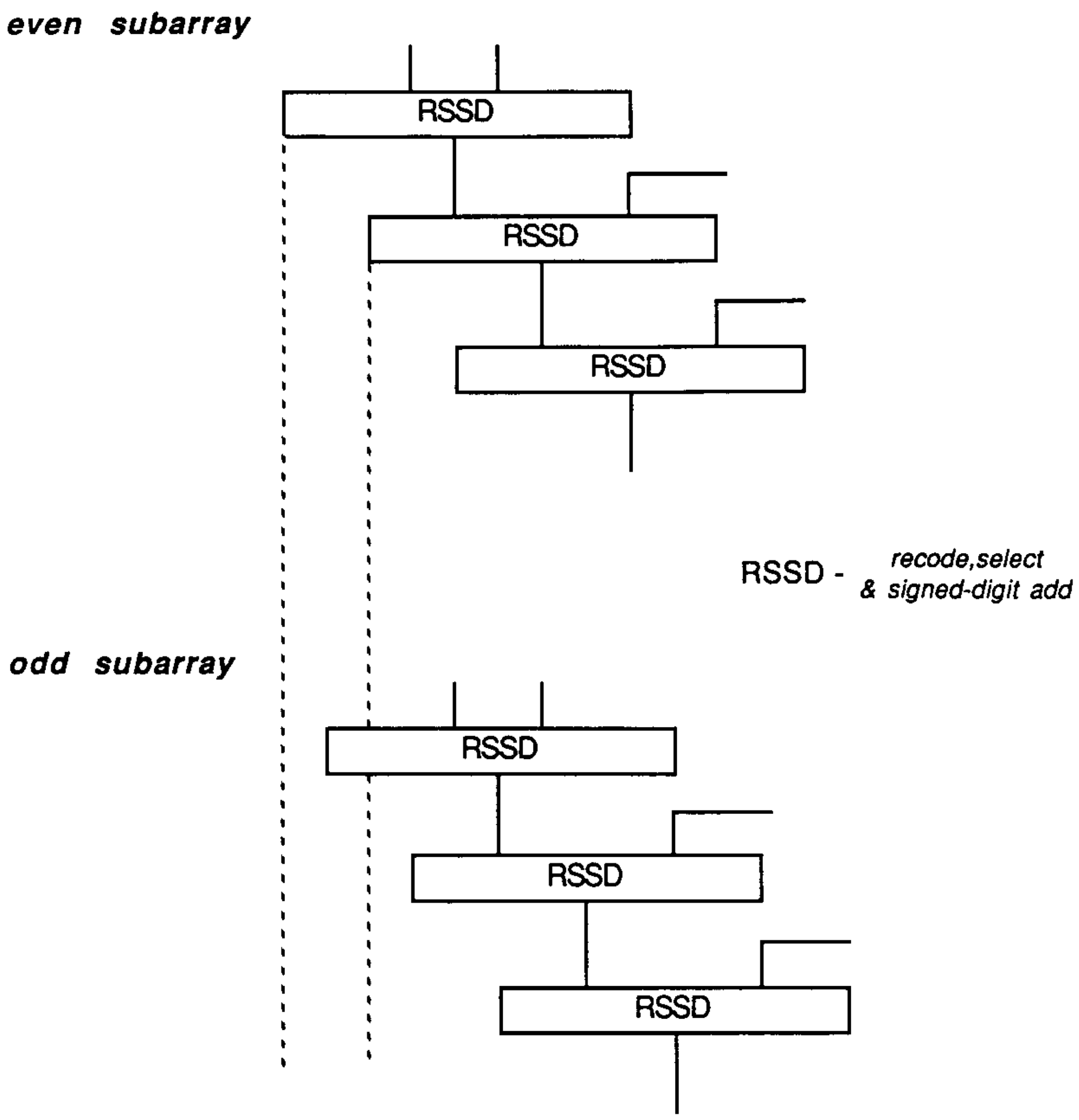


Figure 9. Even and Odd Subarrays

$$|Z_j(even)| \le 18 \tag{5.3}$$

$$|Z_i(odd)| \le 24$$

Next, we produce a composite $Z$ as

$$Z_j = Z_j(even) + Z_{j+1}(odd) \quad |Z_j| \le 42 \tag{5.4}$$

This $Z_j$ is now recoded into $t_{j-1}$ and $s_j$, so that the computation of $P$ can proceed without carry propagation. Since at each step we compute a radix-16 digit, the recoding satisfies

$$Z_j = 16t_{j-1} + s_j \quad |t_{j-1}| \le 2 \quad |s_j| \le 13$$

That is, $P_j = s_j + t_j$ is in the range {-15,...,15}. The resulting recoding is described in Table 5.1.

Table 5.1: Odd-Even Recoding

| Condition | $t_{j-1}$ | $s_j$ |
|---|---|---|
| $\|Z_j\| mod\, 16 < 14$ | $integer(Z_j/16)$ | $remainder(Z_j/16)$ |
| $\|Z_j\| mod\, 16 \ge 14$ **and** $Z_j > 0$ | $integer(Z_j/16) + 1$ | $remainder(Z_j/16) - 16$ |
| $\|Z_j\| mod\, 16 \ge 14$ **and** $Z_j < 0$ | $integer(Z_j/16) - 1$ | $remainder(Z_j/16) + 16$ |

The computation of $P_j$ is shown in Figure 10. The on-the-fly conversion to conventional representation is performed by a radix-16 version of the implementation presented in Section 3.

## 6. Summary

We have reported a multiplication scheme (LRCF) that eliminates the need for a carry-propagate adder. The scheme performs the multiplication most-significant bit first and produces a conventional sign-and-magnitude product (most significant half) by means of an on-the-fly conversion, performed concurrently with the generation of accumulated (redundant) partial products. The scheme is presented for general radix $r$ and radix-4 implementations are described. Two implementations using a linear array of redundant adders are presented, one using signed-digit adders and the other with the carry-save variety. The decision on which one to use depends on technological constraints. We also present an implementation that improves the speed by computing odd and even partial products concurrently.

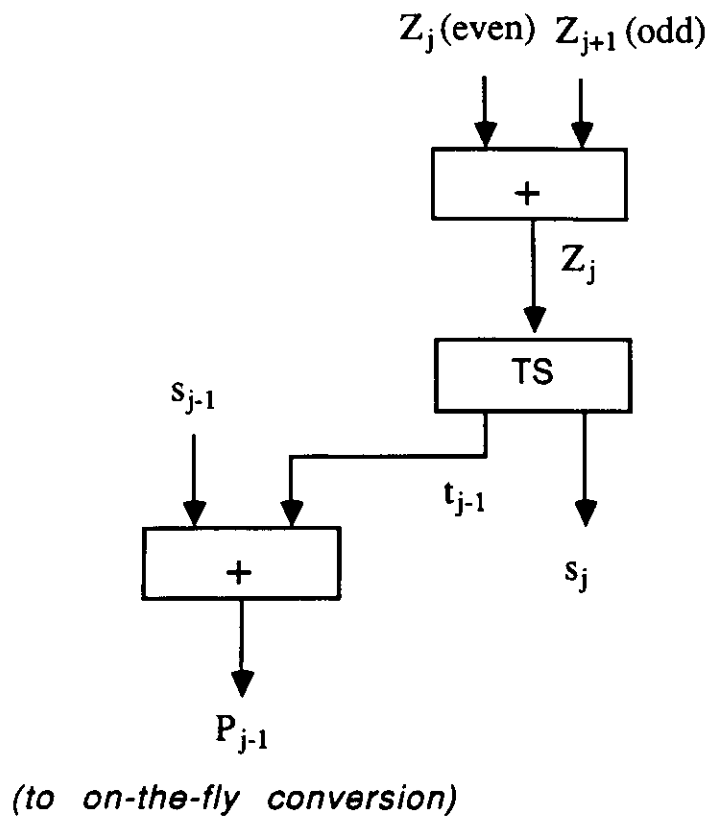


Figure 10. Computation of $P_j$ in even-odd case

We estimate that, for a multiplier of 64 bits, the scheme we described produces a reduction of about 10 gate levels with respect to a conventional scheme using a carry-lookahead adder.

We performed an error analysis of the implementations and indicated a way of implementing the rounding schemes included in the IEEE standard. We conclude that some additional hardware is required for the implementation of these rounding methods. This hardware is not necessary in applications that allow a somewhat larger error than that specified by these rounding schemes.

## References

1. K. Hwang, *Computer Arithmetic*, John Wiley and Sons, 1978.

2. M. Uya, K. Kaneko, and J. Yasui, "A CMOS Floating-Point Multiplier", IEEE Journal of Solid-State Circuits, Vol. SC-19, No.5, October 1984, pp. 697-701.

3. A. Avizienis, "On a flexible implementation of digital computer arithmetic," Information Processing 1962, C.M. Popplewell, Ed., North Holland, 1963, pp. 664-670.

4. A.D. Booth, "A signed binary multiplication technique," Quart. Journal Mech. and Appl. Math., Vol. 4, Part 2, 1951, pp. 236-240.

5. M.D. Ercegovac and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations", IEEE Transactions on Computers, Vol.C-36, No. 7, July 1987, pp.895-897.

6. M.D. Ercegovac and T. Lang, "Fast Multiplication without Carry-propagate Addition, UCLA Computer Science Department Report, 1986.

7. A. Avizienis, "Signed-Digit Number Representation for Fast Parallel Arithmetic", IEEE Transactions Electronic Computers, Vol. EC-10, September 1961, pp. 389-400.

8. J.T. Coonen, "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," Computer, January 1980, pp. 68-79

9. Annon., "Cray X-MP Computer Systems", Four-Processor Mainframe Reference Manual, HR-0097, Cray Research, Inc., 1985.

10. M.D. Ercegovac and T. Lang, "Alternative On-the-Fly Conversion of Redundant into Conventional Representations," UCLA Computer Science Department Report No. CSD-860027, Nov. 1986.

11. J. Iwamura et al., "A 16-bit CMOS/SOS Multiplier-Accumulator," Proc. ICCC82, pp. 151-154, 1982.

12. S. Kuninobu et al., "Design of High-speed MOS Multiplier and Divider Using Redundant Binary Representation," Proc. 8th. Symposium on Computer Arithmetic, 1987, pp. 80-86.

13. Y. Harata et al., "High-Speed Multiplier Using a Redundant Binary Adder Tree," 1984 IEEE International Conference on Computer Design, 1984, pp. 165-170.

14. J.E. Robertson, "A Systematic Approach to the Design of Structures for Arithmetic", Proc. 5th Symposium on Computer Arithmetic, 1981.

## Appendix A: Bit-Level Implementation of Signed-Digit LRCF Scheme

We now discuss a bit-level implementation of some of the modules, to convey an idea of the complexity of the implementation.

(1) *Binary signed-digit adder*

The radix-4 LRCF scheme, described in Section 3, leaves the choice of signed-digit adder open. We implement the adder array using radix-2 signed-digit modules because of their simpler implementation. Several designs of signed-digit adders have been presented in the literature [12, 13, 14]. The particular signed-digit adders used in the multiplication array are quite simple since one of the inputs (the multiple of the multiplicand) is in conventional representation. A suitable implementation, described in [12], is shown in Figure A1. It uses the following code for signed-bits: -1 = 11, 0 = 10, and 1 = 01.

(2) *Generation of $t_{j-1}$ and $s_j$*

The variables $t_{j-1}$ and $s_j$ are function of $Z_j$, as indicated before. The variable $Z_j$ has three radix-2 signed digits, represented by six binary variables $(\zeta_2,z_2)$, $(\zeta_1,z_1)$, $(\zeta_0,z_0)$, where $\zeta_i$ and $z_i$ denote *sign and magnitude* according to the code mentioned above.

The transfer signed-bit $t_{j-1}$ is represented by two binary variables, $\tau$ for sign and $t$ for magnitude. Similarly, $s$ is represented by the binary variables $(\sigma_1,s_1)$ and $(\sigma_0,s_0)$. The switching expressions for these variables are obtained from the following table.

| $Z_j$ | $\zeta_2 z_2$ | $\zeta_1 z_1$ | $\zeta_0 z_0$ | $t_{j-1}$ | $\tau t$ | $s_j$ | $\sigma_1 s_1$ | $\sigma_0 s_0$ |
|---|---|---|---|---|---|---|---|---|
| 5 | 01 | 10 | 01 | 1 | 01 | 1 | 10 | 01 |
| 4 | 01 | 10 | 10 | 1 | 01 | 0 | 10 | 10 |
| 3 | 10 | 01 | 01 | 1 | 01 | -1 | 10 | 11 |
| 2 | 10 | 01 | 10 | 0 | 10 | 2 | 01 | 10 |
| 1 | 10 | 10 | 01 | 0 | 10 | 1 | 10 | 01 |
| 0 | 10 | 10 | 10 | 0 | 10 | 0 | 10 | 10 |
| -1 | 10 | 10 | 11 | 0 | 10 | -1 | 10 | 11 |
| -2 | 10 | 11 | 10 | 0 | 10 | -2 | 11 | 10 |
| -3 | 10 | 11 | 11 | -1 | 11 | 1 | 10 | 01 |
| -4 | 11 | 10 | 10 | -1 | 11 | 0 | 10 | 10 |
| -5 | 11 | 10 | 11 | -1 | 11 | -1 | 10 | 11 |

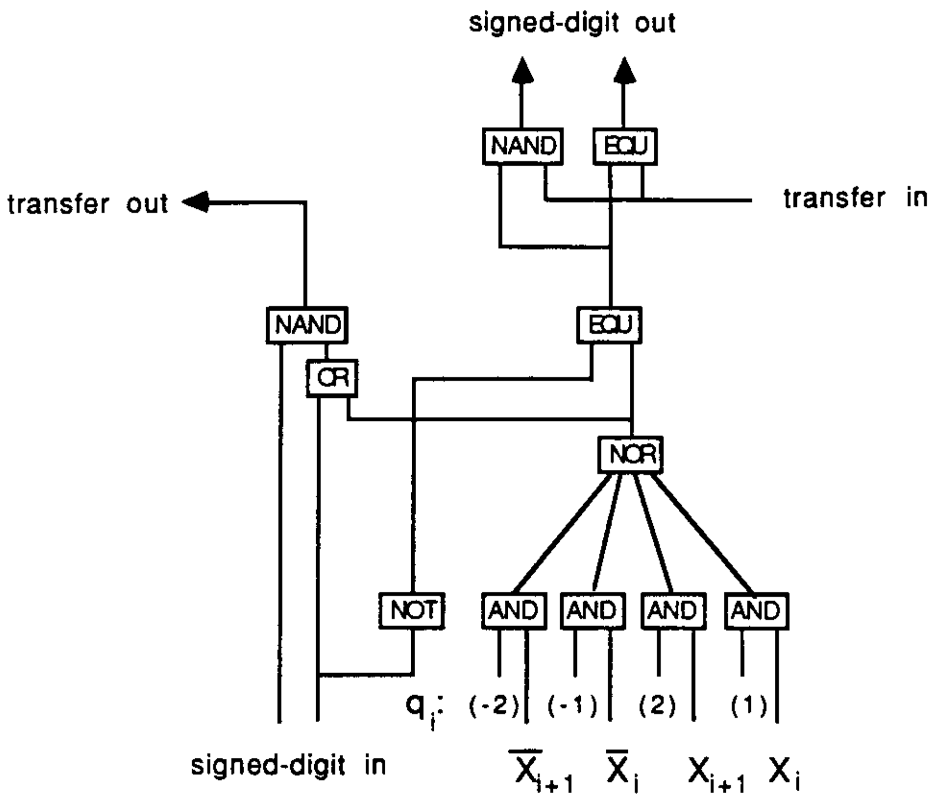


Figure A1: Binary Signed-Digit Adder

From the table we get the following high-level expressions:

$$(\tau,t)=\begin{cases}(\zeta_2,z_2) & \textbf{if} \quad |Z_j| \neq 3\\ (0,1) & \textbf{if} \quad Z_j=3\\ (1,1) & \textbf{if} \quad Z_j=-3\end{cases}$$

$$(\sigma_1,s_1,\sigma_0,s_0)=\begin{cases}(\zeta_1,z_1,\zeta_0,z_0) & \textbf{if} \quad |Z_j| \neq 3\\ (1,0,1,1) & \textbf{if} \quad Z_j=3\\ (1,0,0,1) & \textbf{if} \quad Z_j=-3\end{cases}$$

The corresponding switching expressions are

$$\tau=\zeta_2\zeta_1'\zeta_0' \quad t=z_2+z_1z_0$$

$$\sigma_1=\zeta_1+z_1z_0 \quad s_1=z_1z_0'$$

$$\sigma_0=\zeta_0+z_1z_0 \quad s_0=z_0$$

These expressions result in simple gate networks.

(3) *Addition of $t_j$ and $s_j$ and the computation of $A_j=(a_1,a_0)$, zero$(P_j)$, and sign$(P_j)$*

The addition of $t_j$ and $s_j$ is performed by a 2-bit signed-digit adder, obtaining $P_j$ represented as $(\pi_1 p_1)$ and $(\pi_0,p_0)$. $A_j=(a_1,a_0)$, *zero*$(P_j)$, and *sign*$(P_j)$ are obtained according to (3.11) and (3.14). We get

$$zero(P_j)=p_1'p_0'$$

$$sign(P_j)=\pi_1+p_1'\pi_0$$

$$a_1=(p_1+p_0)(p_1+\pi_0)(p_0'+p_1'+\pi_0')$$

$$a_0=p_0$$

(4) *Conversion control signals*

The conversion control signals, defined in Table 3.1 and (3.13), are implemented assuming the following code for $D_k[j]$:

| $D_k[j]$ | $\delta_1[j]$ | $\delta_0[j]$ |
|---|---|---|
| $u$ | 1 | 0 |
| $n$ | 0 | 0 |
| $d$ | 0 | 1 |

(the subscript $k$ is dropped from $\delta$'s for simplicity)

The resulting switching expressions are simple:

$$\delta_1[j+1] = \delta_1[j] \cdot zero\,(P_j)$$

$$\delta_0[j+1] = \delta_0[j] + \delta_1[j] \cdot sign\,(P_j)$$

with the initial conditions $\delta_1 = 1$ and $\delta_0 = 0$.

(5) *Decrementers*

Since $A_k$ is in the set $\{0,1,2,3\}$, the decrementation required when $D_k = d$ can be performed by a simple network. The final digit $M_k$, represented by two binary variables $m_1$ and $m_0$ (again, we omit the index $k$ out of sympathy for the tired reader), is defined by the following switching expressions:

$$m_1 = (a_1 \oplus a_0)'\delta_0 + a_1\delta_0' = a_1a_0 + a_1\delta_0' + a_1'a_0'\delta_0$$

$$m_0 = a_0 \oplus \delta_0$$

## Appendix B: Bit-Level Implementation of Carry-Save LRCF Scheme

The implementation of the TSA block consists of the following modules:

(1) *A two-bit adder*. This is a standard adder that has inputs $PS_{-1}[j-1]$, $PC_{-1}[j-1]$, $PS_0[j-1]$, $PC_0[j-1]$, $PC_0[j]$ and produces $(U_2, U_1, U_0)$.

(2) *Module T*: a code converter for $t$. This is used to simplify the implementation of the A block. The code for $t$ used is

| $t$ | $t_1 t_0$ |
|---|---|
| 0 | 00 |
| 1 | -1 |
| 2 | 10 |

To obtain this code, the two most significant bits of the output of the two-bit adder $(U_2 U_1)$ produce $(t_1 t_0)$ as follows:

$$t_1 = U_2$$

$$t_0 = U_2 \, XOR \; U_1$$

(3) *Module S*: subtracts $sign(Y_j)$ from $s_j^f$ $(U_1 U_0)$ and produces $s = s_2 s_1 s_0$ in two's complement representation. Its implementation is described by the following table, which indicates the products required and the functions in which these products are used.

| $U_1 U_0$ *sign* | |
|---|---|
| 00 1 | $s_1$ |
| -1 0 | $s_0$ |
| -0 1 | $s_2, s_0$ |
| 1- 0 | $s_2, s_1$ |
| 11 - | $s_2, s_1$ |

The values of $s$ and $t$ are then used as in the implementation described in Section 3 to compute $A_k$ and $zero(P_j)$ and $sign(P_j)$. The corresponding block A implements the following table:

| $t_j$ | $s_j$ | $P_j$ | $a_1a_0$ | $sign(P_j)$ | $zero(P_j)$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 00 | 0 | 1 |
| 0 | 1 | 1 | 01 | 0 | 0 |
| 0 | -1 | -1 | 11 | 1 | 0 |
| 0 | -2 | -2 | 10 | 1 | 0 |
| 0 | -3 | -3 | 01 | 1 | 0 |
| 1 | 0 | 1 | 01 | 0 | 0 |
| 1 | 1 | 2 | 10 | 0 | 0 |
| 1 | -1 | 0 | 00 | 0 | 1 |
| 1 | -2 | -1 | 11 | 1 | 0 |
| 1 | -3 | -2 | 10 | 1 | 0 |
| 2 | 0 | 2 | 10 | 0 | 0 |
| 2 | 1 | 3 | 11 | 0 | 0 |
| 2 | -1 | 1 | 01 | 0 | 0 |
| 2 | -2 | 0 | 00 | 0 | 1 |
| 2 | -3 | -1 | 11 | 1 | 0 |

Using the codes for $s$ and $t$ indicated before, block A is described by the following table (which shows the products required and the functions that use them):

| $t_1t_0s_2s_1s_0$ | -11-0 | --10- | 10-0- | -1-01 | 00-1- | -1--0 | -0--1 |
|---|---|---|---|---|---|---|---|
| $a_1$ | x | | x | x | x | | |
| $a_0$ | | | | | | x | x |
| $sign$ | x | x | | | x | | |
| $zero'$ | | | x | x | x | x | x |

The rest of the conversion to conventional representation is identical to that presented in Section 3.