# Decimal Multiplication With
# Efficient Partial Product Generation

Mark A. Erle, Eric M. Schwarz
Server & Technology Group
International Business Machines
Poughkeepsie, NY 12601, USA

merle@us.ibm.com, eschwarz@us.ibm.com

Michael J. Schulte
Dept. of Electrical & Computer Engr.
University of Wisconsin - Madison
Madison, WI 53706, USA

schulte@engr.wisc.edu

## Abstract

*Decimal multiplication is important in many commercial applications including financial analysis, banking, tax calculation, currency conversion, insurance, and accounting. This paper presents a novel design for fixed-point decimal multiplication that utilizes a simple recoding scheme to produce signed-magnitude representations of the operands thereby greatly simplifying the process of generating partial products for each multiplier digit. The partial products are generated using a digit-by-digit multiplier on a word-by-digit basis, first in a signed-digit form with two digits per position, and then combined via a combinational circuit. As the signed-digit partial products are developed one at a time while traversing the recoded multiplier operand from the least significant digit to the most significant digit, each partial product is added along with the accumulated sum of previous partial products via a signed-digit adder. This work is significantly different from other work employing digit-by-digit multipliers due to the efficiency gained by restricting the range of digits throughout the multiplication process.*

## 1. Introduction

Current floating-point units are typically binary based, not decimal based, for largely two reasons. Binary data can be stored efficiently and manipulated very quickly on digital computers [1]. However, there are compelling reasons to consider base ten for floating-point arithmetic, particularly for business computations. These include: the inexact mapping between some decimal and binary values, a preponderance of business data input, stored, and output in decimal format [2], and humanity's natural* affinity for dec-

---
*Human beings have ten fingers.

imal arithmetic [3]. In fact, due to the importance of decimal arithmetic in commercial applications, specifications for it have been added to the draft revision of the IEEE Standard for Floating-Point Arithmetic [4]. These specifications are more comprehensive than the IEEE Standard for Radix-Independent Floating-Point Arithmetic [5], including formats for single, double, and quadruple precision decimal floating-point numbers.

With the cost of die space continually dropping and the significant speedup achievable in hardware [6], a dedicated decimal floating-point hardware implementation is likely to be considered by microprocessor manufacturers. A fundamental operation for any hardware implementation of decimal arithmetic is multiplication, which is integral to the decimal-dominant applications found in financial analysis, banking, tax calculation, currency conversion, insurance, and accounting. This paper proposes a design for fixed-point Binary Coded Decimal (BCD) multiplication that can be extended to support these applications in compliance with a prevailing decimal arithmetic specification [7] and the IEEE Standard for Floating-Point Arithmetic, currently under revision [4].

Decimal multiplication is much more complicated than binary multiplication due to the need for a greater number of multiplicand multiples and the inefficiency of representing decimal values with two-state devices. Both of these issues complicate partial product generation, while the latter issue complicates partial product accumulation. The algorithm presented in this paper reduces the complexity of partial product generation in a novel way by employing a recoding scheme to restrict the magnitude range of the operand digits. Further, by restricting the range of each digit in the partial product, the complexity of partial product accumulation is also significantly reduced. The algorithm extends previous techniques used for decimal multiplication including the use of lookup tables to produce partial products [10, 11] and the use of signed-digit addition for the accumulation of

the partial products [12].

The outline of the paper is as follows. Section 2 presents related research on decimal multiplication. Sections 3 through 5 contain descriptions of core portions of the algorithm: recode of the operands, generation of the partial products, accumulation of the partial products, and generation of the final product. Section 6 presents a multiplier implementation using the presented scheme along with some implementation options. Section 7 contains a summary of the paper.

## 2. Related Work

Several existing designs for decimal multiplication generate and store multiples of the multiplicand a priori [8, 9], and then use the multiplier digits to select the appropriate multiple as the partial product. An alternative approach is to generate the partial product as needed. Generating the partial products as needed is an ideal approach for three reasons. First, it eliminates the cycles needed to generate the multiples of the multiplicand prior to the start of partial product accumulation. Second, it reduces wiring by eliminating the need to distribute all the multiples to multiplexors controlled by the multiplier digits. And third, in most environments, it eliminates the registers needed to store the multiples. However, these benefits come at a cost in delay to generate the partial products.

The following designs generate the partial products as needed. In [10], Larson describes a digit-by-digit lookup table scheme. The multiplier operand is traversed from least significant digit (LSD) to most significant digit (MSD) and a partial product is generated for each digit in the multiplier operand. The partial product is added along with the previous iteration's properly shifted intermediate product via a carry-propagate adder. In [13], Larson presents a second, faster implementation which employs the lookup scheme just described, but replaces the carry-propagate adder with a four-input carry-save adder. In [11], Ueda presents a lookup table which accepts digits from each operand and carries from adjacent lookup tables. All of these schemes, and similar digit-by-digit lookup table schemes, require significant circuitry and delay to generate the digit-by-digit products, since each digit ranges from zero to nine.

## 3. Recoding of Operands

Throughout this paper, upper case variables denote multiple digit words, lower case variables with subscripts denote decimal digits, and lower case variables with subscripts and indices denote bits. Thus, $a_i$ corresponds to digit $i$ of operand $A$, and $a_i[j]$ corresponds to bit $j$ in digit $i$. Upper case variables with subscripts denote multiple digit words

that are part of an iterative equation, and upper case variables with subscripts and indices denote digits. For example, $IP_i[3]$ corresponds to the fourth LSD in the intermediate product after $i$ iterations. Superscripts are used to differentiate various forms of the same variable. Bits and digits are indexed from least significant to most significant, starting with index zero. Logic equations are shown in a form which lends itself to implementation in CMOS technologies. A subscript next to a constant indicates the base. Last, a decimal number with a line over it indicates the additive inverse of the number. For example, $\overline{6_{10}} = -6_{10}$.

As stated in Section 1, restricting the range of the operand digits leads to a faster generation of a smaller number of partial products. A range which is close to the minimum, yet balanced so as to simplify the recoding, is $-5_{10}$ through $+5_{10}$. Since the magnitude of a product is independent of the sign of the multiplicand and multiplier inputs, this range significantly reduces the combinations of inputs needing to be multiplied. Table 1 shows the reduction in input combinations and complexity achievable by restricting the range of inputs for which digit-by-digit products must be generated. For example, with digit ranges from $0_{10}$ through $9_{10}$, there are a total of $100$ input combinations which can result in 37 unique products. Computing this product set requires 62 minterms with the worst-case output bit using 23 minterms or 19 gate levels.

Using signed-digits, a digit can be equivalently represented by replacing it with the additive inverse of its radix complement and incrementing its next more significant digit [14]. In general, each digit greater than or equal to six must be recoded. However, since a digit can be incremented due to the value of the next less significant digit, the chosen strategy is to evaluate and recode all digits greater than or equal to five. By doing so, the recoding of the digits can occur in parallel as an increment of the next more significant digit will never propagate. Although in the circumstance of a digit being equal to five and its next less significant digit being less than six, the digit need not be recoded, the chosen approach minimizes hardware as only one condition, greater than or equal to five, must be evaluated for each digit position. Figure 1, referred to throughout this paper, provides two examples of three-digit numbers recoded in the range of $-5_{10}$ through $+5_{10}$. The number on line 1 (339) is recoded into the number on line 3 ($34\overline{1}$), and the number on line 2 (265) is recoded into $3\overline{3}\overline{5}$, the digits of which can be found on lines 16, 9, and 4, respectively.

To restrict the range of the operand digits to $-5_{10}$ through $+5_{10}$, the multiplicand is sent to a set of $n$ recoders, where $n$ is the number of digits of the operands, and each multiplier digit is sent to a single recoder, as it is being used. Each recoder block receives as input one four-bit BCD operand digit, $a_i$, and a single bit, $ge5_{i-1}$, indicating if the next less significant digit is greater than or equal to five and produces as output a four-bit signed-magnitude digit,

**Table 1. Complexity of Digit-by-Digit Products for Different Ranges of Decimal Inputs.**

| range of inputs | input combinations | unique products | total minterms† | maximum minterms per output† | maximum gate levels per output‡ |
|---|---|---|---|---|---|
| $[0-9]$ x $[0-9]$ | 100 | 37 | 62 | 23 | 19 |
| $[1-9]$ x $[1-9]$ | 81 | 36 | 61 | 21 | 17 |
| $[2-9]$ x $[2-9]$ | 64 | 30 | 55 | 20 | 16 |
| $[0-5]$ x $[0-5]$ | 36 | 15 | 20 | 7 | 8 |
| $[1-5]$ x $[1-5]$ | 25 | 14 | 20 | 7 | 7 |
| $[2-5]$ x $[2-5]$ | 16 | 10 | 15 | 5 | 6 |

† Espresso results in sum-of-products form

‡ SIS results with library of INV, NAND2, NAND3, NAND4, NOR2, NOR3, AOI21, AOI22, XOR, and XNOR cells

$a_i^S$, and a single bit, $ge5_i$, indicating if the current digit is greater than or equal to five. The superscript $S$ indicates the result of the recoding is a signed-magnitude digit. Figure 2 shows a block diagram of a recoder, and Equation 1 describes its function as a collection of sub-functions selected by specific classifications of the input data. Although the equations in this section are shown based on digits of the multiplicand operand $A$, the same equations are applicable to the digits of multiplier operand $B$.

$$a_i^S = \begin{cases} a_i & \text{if } a_i < 5_{10} \ \& \ a_{i-1} < 5_{10} \\ a_i^I = a_i + 1 & \text{if } a_i < 5_{10} \ \& \ a_{i-1} \geq 5_{10} \\ a_i^C = -(10_{10} - a_i) & \text{if } a_i \geq 5_{10} \ \& \ a_{i-1} < 5_{10} \\ a_i^{IC} = -(9_{10} - a_i) & \text{if } a_i \geq 5_{10} \ \& \ a_{i-1} \geq 5_{10} \end{cases} \quad (1)$$

The last three sub-functions are increment, complement, and increment & complement, respectively, hence the superscripts. The circuit implementations for each of these sub-functions are simplified based on the limited range of their inputs. That is, increment only occurs on values zero through four ($a_i < 5_{10}$), and both complement and increment & complement only occur on values five through nine ($a_i \geq 5_{10}$).

The following sets of equations describe the logic of these three sub-functions. In each four-bit signed-magnitude digit, bit [3] represents the sign, and bits [2:0] represent the magnitude. Only Equations 2 through 8 are unique and require circuitry. The different forms of the operand digit, along with the unaltered operand digit are input to multiplexor logic that selects the correct digit based on $ge5_i$ and $ge5_{i-1}$ (Equation 8).

$$a_i^I[3] = 0$$
$$a_i^I[2] = \overline{a_i[2] \cdot (a_i[1] + a_i[0])} \quad (2)$$
$$a_i^I[1] = a_i[1] \oplus a_i[0] \quad (3)$$
$$a_i^I[0] = \overline{a_i[0]} \quad (4)$$

$$a_i^C[3] = 1$$
$$a_i^C[2] = \overline{\overline{a_i[2]} + (a_i[1] \cdot a_i[0])} \quad (5)$$
$$a_i^C[1] = \overline{a_i[1] \oplus a_i[0]} \quad (6)$$
$$a_i^C[0] = a_i[0]$$

$$a_i^{IC}[3] = 1$$
$$a_i^{IC}[2] = \overline{a_i[3] + a_i[1]} \quad (7)$$
$$a_i^{IC}[1] = a_i[1]$$
$$a_i^{IC}[0] = \overline{a_i[0]}$$

$$ge5_i = \overline{\overline{a_i[3] \cdot a_i[2] \cdot (a_i[1] + a_i[0])}} \quad (8)$$

In the case of recoding the multiplicand operand $A$, the $n^{th}$ digit needs to be set to $1_{10}$ if the MSD is greater than or equal to five (i.e., when $ge5_{n-1}$ is high). This can be realized by concatenating $ge5_{n-1}$ with three leading zeros. The recoded multiplicand operand $A^S$ and a digit from the recoded multiplier operand, $b_i^S$, are input to digit multiplier blocks described in the next section to generate a partial product $P_i^O$ in overlapped form.

## 4. Word-by-Digit Partial Product Generation

To reduce the area and delay of generating partial products, the range of the input digits for which digit-by-digit products must be generated is restricted in three ways. The first restriction sets an upper bound on the input digits by recoding the operands into signed-magnitude digits with a range of $-5_{10}$ to $+5_{10}$, as described in Section 3. The second restriction sets a limit on the possible input digit combinations by applying the principle that the absolute value

| line | cycle | function (*line # or "value"*) | example § |
|---|---|---|---|
| 1 | 0 | latch multiplicand | 3 3 9 |
| 2 |  | latch multiplier | 2 6 5 |
| 3 | 1 | recode multiplicand (1) | $3\,4\,\bar{1}$ |
| 4 |  | recode multiplier digit [0] (2) | $\bar{5}$ |
| 5 |  | generate partial product | $\bar{5}\,\bar{0}\,5$ |
| 6 |  | in overlapped form (3,4) | $\bar{1}\,\bar{2}\,0$ |
| 7 | 2 | convert partial product to | $\bar{2}\,3\,0\,5$ |
| 8 |  | non-overlapped form (5,6) |  |
| 9 |  | recode multiplier digit [1] (2) | $\bar{3}$ |
| 10 |  | generate partial product | $1\,\bar{2}\,3$ |
| 11 |  | in overlapped form (3,9) | $\bar{1}\,\bar{1}\,\bar{0}$ |
| 12 | 3 | add partial product (7) to | $\bar{2}\,3\,0\,5$ |
| 13 |  | intermediate product ("0"); |  |
| 14 |  | convert partial product to | $\bar{1}\,0\,\bar{2}\,3$ |
| 15 |  | non-overlapped form (10,11) |  |
| 16 |  | recode multiplier digit [2] (2) | $\bar{3}$ |
| 17 |  | generate partial product | $\bar{1}\,2\,\bar{3}$ |
| 18 |  | in overlapped form (3,16) | $1\,1\,\bar{0}$ |
| 19 | 4 | convert LSD of intermediate | $\underline{\underline{5}}$ |
| 20 |  | product (12); transfer out | 0 |
| 21 |  | add partial product (14) to | $\bar{1}\,\bar{2}\,1\,3$ |
| 22 |  | intermediate product (12,20) |  |
| 23 |  | convert partial product to | $1\,0\,2\,\bar{3}$ |
| 24 |  | non-overlapped form (17,18) |  |
| 25 | 5 | convert LSD of intermediate | $\underline{\underline{3}}$ |
| 26 |  | product (21); transfer out | 0 |
| 27 |  | add partial product (23) to | $1\,\bar{1}\,0\,\bar{2}$ |
| 28 |  | intermediate product (21,26) |  |
| 29 | 6 | first half of conversion |  |
| 30 |  | to BCD (27) |  |
| 31 |  | convert LSD of intermediate | $\underline{\underline{8}}$ |
| 32 |  | product (27); transfer out | $\bar{1}$ |
| 33 | 7 | second half of conversion | $\underline{\underline{0}}\,\underline{\underline{8}}\,\underline{\underline{9}}$ |
| 34 |  | to BCD (27,32) |  |

§ Digits in the final product are double underlined.

**Figure 1. Algorithm Example.**



**Figure 2. Recoder Block: (a) Single Digit, (b) n-Digit Operand.**

of a product is independent of the sign of the input digits. The third restriction sets a lower bound on the input digits by applying the observation that if either digit is zero, the product is zero, and if either digit is one, the product is the other digit.

With these three restrictions on the input digits, the range is reduced to only $2_{10}$ through $5_{10}$ when computing a product. Thus, only $16_{10}$ combinations of the inputs are possible resulting in ten different products with a range of $4_{10}$ through $25_{10}$ (hence the need for a two-digit product). With existing schemes, the range of digits is $0_{10}$ through $9_{10}$, which yields $100_{10}$ possible combinations of the two inputs.
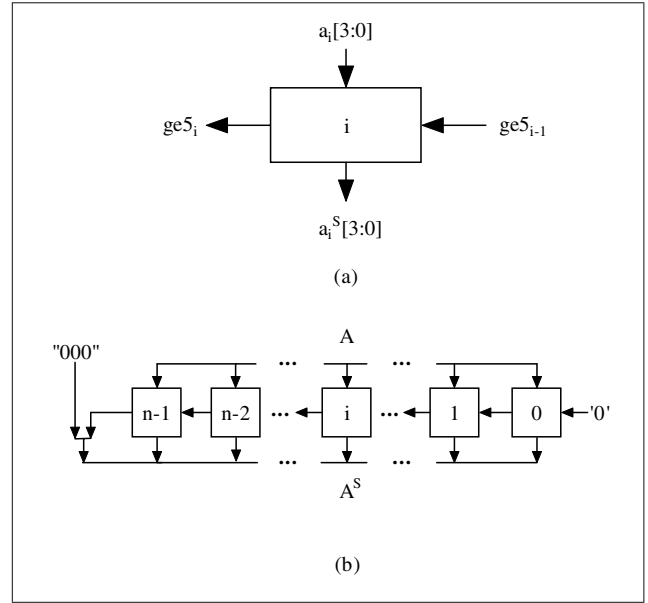
Table 1 illustrates for various input ranges the significant reduction in complexity achievable by restricting the number of input combinations.

To generate a partial product on a word-by-digit basis, the recoded multiplicand and a recoded digit from the multiplier are input to $n + 1$ digit multiplier blocks (see Figure 3b). Note since the $n^{th}$ digit of the recoded multiplicand has at most a magnitude of $1_{10}$, the digit multiplier block in this position can be replaced with a simpler circuit to produce either $0_{10}$ or the recoded multiplier digit, $|b_i^S|$. Each multiplier block receives as input two, four-bit, signed-magnitude digits, $a_i^S$ and $b_i^S$, and produces as output two, signed-magnitude partial product digits, $p_{i+1}^O$ and $p_i^O$. The superscript $O$ indicates the partial product is in an overlapped form since each digit multiplier block yields two digits. Equation 9 describes the function to generate a digit-by-digit product, in absolute-value form, as a collection of sub-functions selected by specific classifications of the input digits. The superscript $T$ indicates the sub-function output is realized via a lookup table or a combinational circuit structure.

To simplify the removal of the overlap in the partial product, the range of $|p_i^T|$ is restricted to $0_{10}$ through $5_{10}$ by again using signed-magnitude digits. With this restriction, which matches the inherent restriction on the other sub-functions in Equation 9, four bits are needed for the product's LSD (range of $-4_{10}$ through $+5_{10}$), and two bits are needed for the product's MSD (range of $0_{10}$ through $2_{10}$). Table 2 shows for inputs ranging from

## Table 2. Restricted-Range, Signed-Magnitude Products.

| x | $2_{10}$ | $3_{10}$ | $4_{10}$ | $5_{10}$ |
|---|---|---|---|---|
| $2_{10}$ | $04_{10}$ $00,0100_2$ | $1\bar{4}_{10}$ $01,1100_2$ | $1\bar{2}_{10}$ $01,1010_2$ | $10_{10}$ $01,0000_2$ |
| $3_{10}$ | $1\bar{4}_{10}$ $01,1100_2$ | $1\bar{1}_{10}$ $01,1001_2$ | $12_{10}$ $01,0010_2$ | $15_{10}$ $01,0101_2$ |
| $4_{10}$ | $1\bar{2}_{10}$ $01,1010_2$ | $12_{10}$ $01,0010_2$ | $2\bar{4}_{10}$ $10,1100_2$ | $20_{10}$ $10,0000_2$ |
| $5_{10}$ | $10_{10}$ $01,0000_2$ | $15_{10}$ $01,0101_2$ | $20_{10}$ $10,0000_2$ | $25_{10}$ $10,0101_2$ |

$2_{10}$ through $5_{10}$ the two-digit, signed-magnitude products conforming to this magnitude restriction. Although the LSD has a negative sign in some instances, the MSD is always positive, and thus the two-digit product is a positive value. Figure 3a shows the block diagram of a digit multiplier block, and Equations 10 - 15 show how the two-digit products are developed.

$$|p_{i+1}^O, p_i^O| =$$

$$\begin{cases} 00,0000 & \text{if } |a_i^S| = 0_{10} \text{ or } |b_i^S| = 0_{10} \\ 00,0b_i^S[2:0] & \text{if } |a_i^S| = 1_{10} \text{ \& } |b_i^S| > 0_{10} \\ 00,0a_i^S[2:0] & \text{if } |a_i^S| > 1_{10} \text{ \& } |b_i^S| = 1_{10} \\ |p_{i+1}^T, p_i^T| & \text{if } |a_i^S| > 1_{10} \text{ \& } |b_i^S| > 1_{10} \end{cases} \quad (9)$$

Since the signs of the recoded operand digits were not considered when generating the digit-by-digit products, the partial product at this point is in absolute-value form. Thus, the sign of the recoded operand digits must be used to convert $|P_i^O|$ into a properly signed partial product. This step is necessary before attempting to add the overlapping portions of the word-by-digit products as not doing so could yield an incorrect partial product. To develop a partial product with the correct sign, $P_i^O$, the exclusive-or (XOR) of the input signs (i.e., $a_i^S[3] \oplus b_i^S[3]$), is used in two places. First, it directly becomes the sign of the product's MSD, $p_{i+1}^O[2]$. Second, it is XORed with the sign of the product's LSD, $|p_i^O[3]|$, to produce $p_i^O[3]$. Figure 1, lines 5/6, 10/11, and 17/18, provide examples of the digit multiplier blocks yielding the sign-corrected partial products in overlapped form.

Ultimately, all the partial products need to be properly aligned with respect to one another and added together. The approach chosen in this work is to iteratively accumulate the partial products via the signed-digit adder described by Svoboda in [12]. Svoboda's adder accepts two uniquely encoded signed-digits (see Table 3) in the range of $-6_{10}$ through $+6_{10}$ and yields a sum in the same range. Note a property of the encoding shown in Table 3 is the additive inverse is obtained by taking the one's complement.
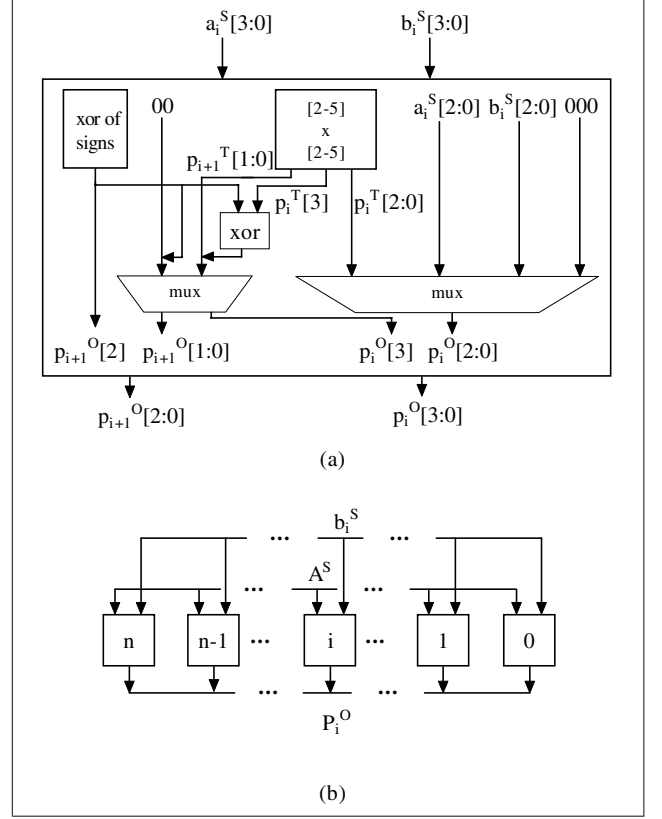


(a)



(b)

**Figure 3. Digit Multiplier Block: (a) Single Digit, (b) n-Digit.**

$$|p_i^T[0]| = \overline{\overline{a_i^S[0] + \overline{b_i^S[0]}}} \quad (10)$$

$$|p_i^T[1]| = \overline{\overline{a_i^S[1] \cdot \overline{b_i^S[1]} \cdot \overline{b_i^S[0]}} \cdot} \quad (11)$$
$$\overline{\overline{a_i^S[1]} \cdot \overline{a_i^S[0]} \cdot b_i^S[1]}$$

$$|p_i^T[2]| = \overline{\overline{a_i^S[0] \cdot b_i^S[2] \cdot b_i^S[0]} \cdot} \quad (12)$$
$$\overline{\overline{(a_i^S[1] \oplus a_i^S[0])} \cdot b_i^S[1] \cdot b_i^S[0]} \cdot$$
$$\overline{\overline{a_i^S[1]} \cdot b_i^S[1] \cdot \overline{b_i^S[0]}} \cdot$$
$$\overline{\overline{a_i^S[1]} \cdot \overline{a_i^S[0]} \cdot \overline{b_i^S[1]} \cdot \overline{b_i^S[0]}}$$

$$|p_i^T[3]| = \overline{\overline{a_i^S[0] \cdot b_i^S[1] \cdot \overline{b_i^S[0]}} \cdot} \quad (13)$$
$$\overline{\overline{a_i^S[1]} \cdot b_i^S[1] \cdot b_i^S[0]} \cdot$$
$$\overline{\overline{a_i^S[1] \oplus a_i^S[0]} \cdot b_i^S[1] \cdot \overline{b_i^S[0]}}$$

$$|p_{i+1}^T[0]| = \overline{\overline{a_i^S[1] \cdot b_i^S[2]} \cdot} \quad (14)$$
$$\overline{\overline{(a_i^S[2] + a_i^S[0])} \cdot b_i^S[1]} \cdot$$
$$\overline{\overline{a_i^S[1]} \cdot b_i^S[0]}$$

$$|p_{i+1}^T[1]| = \overline{\overline{a_i^S[2] + \overline{b_i^S[2]}}} \quad (15)$$

## Table 3. Svoboda Signed-Digit Code [12].

| decimal | encoded digit | | | | |
|---------|-----|-----|-----|-----|-----|
| value | [4] | [3] | [2] | [1] | [0] |
| 6 | 1 | 0 | 0 | 1 | 0 |
| 5 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| $\bar{0}$ | 1 | 1 | 1 | 1 | 1 |
| $\bar{1}$ | 1 | 1 | 1 | 0 | 0 |
| $\bar{2}$ | 1 | 1 | 0 | 0 | 1 |
| $\bar{3}$ | 1 | 0 | 1 | 1 | 0 |
| $\bar{4}$ | 1 | 0 | 0 | 1 | 1 |
| $\bar{5}$ | 1 | 0 | 0 | 0 | 0 |
| $\bar{6}$ | 0 | 1 | 1 | 0 | 1 |

Recall the partial product at this point is properly signed but still in an overlapped form. Each digit position[†] has one four-bit, signed-magnitude digit whose range is $-5_{10}$ through $+5_{10}$ and one three-bit, signed-magnitude digit whose range is $-2_{10}$ through $+2_{10}$. The sums for these ranges of overlapping signed-digits, suitable for entry into a Svoboda adder, are in bold type in Table 4. In each entry of this table, the digit on the right is a sum digit in position $i$, and the digit on the left is a transfer digit, which is added to the sum digit in position $i+1$. The term transfer digit is used because it is used to indicate when a carry or a borrow occurs. To achieve the desired encoding, a combinatorial circuit is needed to recode the signed-magnitude digits in the partial product $P_i^O$ into signed-digits ($P_i$). A straightforward implementation of this recoding step requires ten logic levels, as determined by SIS. The recoded partial product, $P_i$, is then added to the intermediate product, $IP_{i-1}$, as described in the next section. Figure 1, lines 7, 14, and 23, provide examples of generating a non-overlapped partial product from the sign-corrected partial products in overlapped form.

## 5. Accumulation of Partial Products and Generation of Final Product

As the recoded multiplier operand is traversed from LSD to MSD, the partial product, $P_i$, needs to be added to the sum of the previous partial products. The accumulated sum of partial products is termed the intermediate product and is designated $IP_i$, where the subscript indicates how many partial products have been accumulated. The accumulation occurs in an iterative manner with the intermediate product

---

[†]The MSD and LSD only have one digit in their position.

being shifted to the right one digit position each iteration to achieve a multiplication of the current partial product by $10_{10}$, thus accounting for the increase in weight of each successive multiplier digit. Each iteration, $n+1$ digits from the partial product, $P_i$, and $n+1$ digits from the intermediate product, $IP_{i-1}$, pass through $n+1$ Svoboda digit adders. The range of inputs and their signed-digit sums are shown in Table 4. Figure 1, lines 12, 21, and 27, provide examples of accumulating the partial products.

In shifting the intermediate product one digit position to the right, the LSD is made available for completion as no subsequent partial product digits will be added to this digit. Since this emergent digit is still in the signed-digit code described in Table 3, it must be converted to BCD. During the conversion process, the transfer digit from the previous iteration's intermediate product LSD, $t_{i-1}$, must be taken into account. Logically, the conversion is as follows. If the LSD is greater than zero, the LSD is simply converted to BCD and then decremented if the input transfer digit is $-1_{10}$. If the LSD is less than or equal to zero, the radix complement of the additive inverse of the LSD is converted to BCD and then decremented if the input transfer digit is $-1_{10}$ (only the least significant four bits are kept). Lastly, an output transfer digit, $t_i$, is assigned a value of $-1_{10}$ if the LSD is negative or if the LSD is $0_{10}$ and the input transfer digit is $-1_{10}$, otherwise it is assigned a value of $0_{10}$. Note since the transfer digit in this situation only indicates a borrow or no borrow, and a single bit can be used. Equations 16 and 17 show the different cases for converting the intermediate product LSD and generating the transfer bit, respectively. A straightforward implementation of this conversion and generation of a transfer bit requires twelve logic levels, as determined by SIS. The final product is identified by $FP$.

$$fp_i = \begin{cases} (IP_i[0] \to BCD) + t_{i-1} & \text{if } IP_i[0] \geq 1_{10} \\ 10 - (\overline{IP_i[0]} \to BCD) + t_{i-1} & \text{if } IP_i[0] \leq 0_{10} \end{cases} \quad (16)$$

$$t_i = \begin{cases} 0_{10} & \text{if } IP_i[0] \geq 1_{10} \\ -1_{10} & \text{if } IP_i[0] \leq 0_{10} \ \& \ t_{i-1} = -1_{10} \end{cases} \quad (17)$$

After all the multiplier digits have been processed, the signed-digit outputs of the Svoboda adders comprising $IP_{n-1}$ need to be converted to BCD to produce the final product digits, $fp_{2n-1}$ to $fp_n$. Additionally, the transfer bit, $t_{n-2}$, must be added to the LSD, i.e., $IP_{n-1}[0]$. The algorithm to convert the signed-digits, which is on the order of carry-propagate addition, is fully described in [12]. Figure 1, lines 29/33, provide an example of converting an intermediate product ($1\bar{1}0$) and a transfer bit ($\bar{1}$) into BCD digits.

## 6. Multiplier Implementation

Figure 4 shows one possible multiplier implementation using the presented ideas. As shown, this implementation

**Table 4. Restricted-Range, Signed-Digit Sums [12] (All Digits Are Decimal).**

| + | $\bar{6}$ | $\bar{5}$ | $\bar{4}$ | $\bar{3}$ | $\bar{2}$ | $\bar{1}$ | $\bar{0}$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | $0\bar{0}$ | $01$ | $02$ | $03$ | $04$ | $05$ | $1\bar{4}$ | $1\bar{4}$ | $1\bar{3}$ | $1\bar{2}$ | $1\bar{1}$ | $1\bar{0}$ | $11$ | $12$ |
| 5 | $0\bar{1}$ | $0\bar{0}$ | $01$ | $02$ | $03$ | $04$ | $05$ | $05$ | $1\bar{4}$ | $1\bar{3}$ | $1\bar{2}$ | $1\bar{1}$ | $1\bar{0}$ | $11$ |
| 4 | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $01$ | $02$ | $03$ | $04$ | $04$ | $05$ | $1\bar{4}$ | $1\bar{3}$ | $1\bar{2}$ | $1\bar{1}$ | $1\bar{0}$ |
| 3 | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $01$ | $02$ | $03$ | $03$ | $04$ | $05$ | $1\bar{4}$ | $1\bar{3}$ | $1\bar{2}$ | $1\bar{1}$ |
| 2 | $0\bar{4}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $01$ | $02$ | $02$ | $03$ | $04$ | $05$ | $1\bar{4}$ | $1\bar{3}$ | $1\bar{2}$ |
| 1 | $\bar{1}5$ | $0\bar{4}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $01$ | $01$ | $02$ | $03$ | $04$ | $05$ | $1\bar{4}$ | $1\bar{3}$ |
| 0 | $\bar{1}4$ | $05$ | $0\bar{4}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $00$ | $01$ | $02$ | $03$ | $04$ | $05$ | $1\bar{4}$ |
| $\bar{0}$ | $\bar{1}4$ | $\bar{1}5$ | $0\bar{4}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $0\bar{0}$ | $01$ | $02$ | $03$ | $04$ | $05$ | $1\bar{4}$ |
| $\bar{1}$ | $\bar{1}3$ | $\bar{1}4$ | $\bar{1}5$ | $0\bar{4}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{1}$ | $0\bar{0}$ | $01$ | $02$ | $03$ | $04$ | $05$ |
| $\bar{2}$ | $\bar{1}2$ | $\bar{1}3$ | $\bar{1}4$ | $\bar{1}5$ | $0\bar{4}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $01$ | $02$ | $03$ | $04$ |
| $\bar{3}$ | $\bar{1}1$ | $\bar{1}2$ | $\bar{1}3$ | $\bar{1}4$ | $\bar{1}5$ | $0\bar{4}$ | $0\bar{3}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $01$ | $02$ | $03$ |
| $\bar{4}$ | $\bar{1}0$ | $\bar{1}1$ | $\bar{1}2$ | $\bar{1}3$ | $\bar{1}4$ | $\bar{1}5$ | $0\bar{4}$ | $0\bar{4}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $01$ | $02$ |
| $\bar{5}$ | $\bar{1}\bar{1}$ | $\bar{1}0$ | $\bar{1}1$ | $\bar{1}2$ | $\bar{1}3$ | $\bar{1}4$ | $\bar{1}5$ | $\bar{1}5$ | $0\bar{4}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ | $01$ |
| $\bar{6}$ | $\bar{1}\bar{2}$ | $\bar{1}\bar{1}$ | $\bar{1}0$ | $\bar{1}1$ | $\bar{1}2$ | $\bar{1}3$ | $\bar{1}4$ | $\bar{1}4$ | $\bar{1}5$ | $0\bar{4}$ | $0\bar{3}$ | $0\bar{2}$ | $0\bar{1}$ | $0\bar{0}$ |

requires $n + 4$ cycles, which is the same latency as the design described in [9]. In the first cycle, operand A and a single digit of operand B are recoded. Then, the outputs of the recoder blocks are input to the digit multipliers to yield a sign-corrected partial product in overlapped form. In the second cycle, the overlap of the two-digit products is removed and the partial product is recoded in a manner appropriate for a Svoboda signed-digit adder. For the next $n$ cycles, a partial product is added to the previous iteration's intermediate product, and a new partial product is generated. In the last two cycles, the final intermediate product is converted into BCD digits.

Figure 1 shows an example of multiplying $339_{10}$ by $265_{10}$ using the proposed multiplier implementation. In cycle 1, the multiplicand and the LSD of the multiplier are recoded as described in Section 3 into the signed-digit numbers $34\bar{1}$ (line 3) and $\bar{5}$ (line 4), respectively. Also in cycle 1, the recoded multiplicand (line 3) is multiplied by the LSD of the recoded multiplier (line 4) as described in Section 4 to yield the partial product in overlapped form (lines 5/6). In cycle 2, the partial product generated in overlapped form in cycle 1 is converted to non-overlapped form (line 7). Additionally, the next more significant digit in the multiplier is recoded (line 9) and a partial product based on this digit is generated in overlapped form (lines 10/11). In cycle 3, the accumulation of the partial product as described in Section 5 is initiated by adding the partial product in line 7 to the intermediate product, previously initialized to zero (line 12). Also in cycle 3, the partial product in overlapped form from the previous cycle is converted to non-overlapped form (line 14), the MSD of the multiplier digit is recoded (line 16), and a partial product based on this digit is generated in overlapped form (lines 17/18). In cycle 4, the first digit of the final product (i.e., the LSD) is produced by converting the LSD of the intermediate product to BCD (line 19). The conversion, described in Section 5, takes into account the previously cleared transfer bit and produces an output transfer bit for the next intermediate product's LSD conversion to BCD (line 20). Also in cycle 4, another partial product is added to the intermediate product (line 21) and the previous cycle's partial product is converted to non-overlapped form (line 23). Cycle 5's function includes the conversion to BCD of the intermediate product LSD developed in cycle 4 (line 25), the generation of an output transfer bit (line 26), and the addition of the partial product developed in cycle 4 to the intermediate product (line 27). In cycle 6, the two-cycle process of converting the final intermediate product to BCD digits is initiated as described in Section 5 (line 29). Also in cycle 6, another intermediate product LSD is converted to BCD (line 31) and an output transfer bit is developed (line 32). In the final cycle, 7, the conversion of the final intermediate product to BCD digits is completed (line 33).

Although the implementation just shown is efficient, in terms of its partial product generation, and has good latency, there is opportunity for further research in recoding the input to the Svoboda adder or performing the iterative addition portion with an alternative approach. An alternative is to move the recoding needed for the Svoboda adder to a point earlier in the algorithm. One option is to emerge from the digit multiplier block in the encoding described in Table 3. Regardless, the partial products are initially produced in an overlapped form and need to be corrected.

The issue of having to recode for the Svoboda adder can be removed by not using a Svoboda adder. Instead, an approach employing decimal counters could be used, similar to those described in [9]. Since the inputs to the counters, the partial product, $P_i$, and intermediate product, $IP_{i-1}$, are in a restricted range, the counters could be simplified. However, this benefit needs to be weighed against the cost

of handling the presence of sign bits in each digit position.

## 7. Summary

A novel approach was described for fixed-point decimal multiplication which utilizes restricted-range, signed-digits throughout the multiplication process to generate and accumulate the partial products in an efficient manner. To achieve the restricted range, a simple recoding scheme was shown to produce signed-magnitude representations of the operands. It was further shown how the partial product generation takes the recoded digits, which are in the range of $-5_{10}$ through $+5_{10}$, and uses simple combinational logic to obtain products for input digits in the range $2_{10}$ through $5_{10}$. The steps necessary to handle the signs of the input operands and detect and handle the cases where either input digit is 0 or 1, were also described. It was then described how the results from the partial product generation logic are recoded and added to the accumulated sum of previous partial products via a signed-digit adder. Original aspects of this work include: 1) the method used for recoding the digits into a signed-magnitude representation; 2) the design of the decimal partial product generation; and, 3) the recoding of the partial products before sending them into the signed-digit adder.

## Acknowledgment

## References

[1] W. S. Brown and P. L. Richman, "The Choice of Base," *Communications of the ACM*, vol. 12, pp. 560–561, October 1969.

[2] A. Tsang and M. Olschanowsky, "A Study of Database 2 Customer Queries," IBM Technical Report 03.413, IBM, San Jose, CA, April 1991.

[3] G. Ifrah, *The Universal History of Computing – From the Abacus to the Quantum Computer*. New York, NY: John Wiley and Sons, Inc., 2001. Translated from the French, and with Notes by E. F. Harding.

[4] IEEE Standards Committee, "IEEE Standard for Floating-Point Arithmetic." World Wide Web. http://754r.ucbtest.org/drafts/754r.–pdf.

[5] Floating-Point Working Group, *ANSI/IEEE Std 854-1987: IEEE Standard for Radix-Independent Floating-Point Arithmetic*. New York: The Institute of Electrical and Electronics Engineers, October 1987. 16 pages.

[6] M. A. Erle, M. J. Schulte, and J. M. Linebarger, "Potential Speedup Using Decimal Floating-Point Hardware," in *Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1073–1077, November 2002.

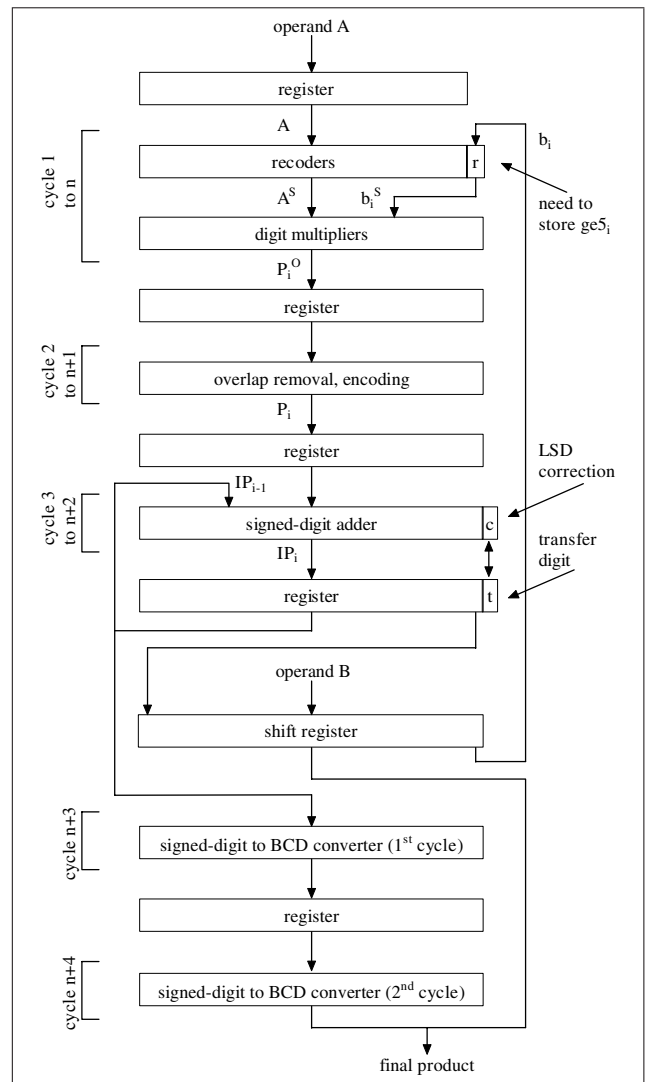[7] M. F. Cowlishaw, "General Decimal Arithmetic Specification." World Wide Web. http://www2.hursley.ibm.com/decimal/decarith.–html.

**Figure 4. Multiplier Implementation.**

[8] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough, "The IBM z900 Decimal Arithmetic Unit," in *Asilomar Conference on Signals, Systems, and Computers*, vol. 2, pp. 1335–1339, November 2001.

[9] M. A. Erle and M. J. Schulte, "Decimal Multiplication Via Carry-Save Addition," in *Conference on Application-Specific Systems, Architectures, and Processors*, pp. 348–358, June 2003.

[10] R. H. Larson, "Medium Speed Multiply," *IBM Technical Disclosure Bulletin*, p. 2055, December 1973.

[11] T. Ueda, "Decimal Multiplying Assembly and Multiply Module," *U.S. Patent #5,379,245*, January 1995.

[12] A. Svoboda, "Decimal Adder with Signed Digit Arithmetic," *IEEE Transaction on Computers*, vol. C, pp. 212–215, March 1969.

[13] R. H. Larson, "High Speed Multiply Using Four Input Carry Save Adder," *IBM Technical Disclosure Bulletin*, pp. 2053–2054, December 1973.

[14] A. Avizienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 389–400, September 1961.