

# Decimal Multiplication Via Carry-Save Addition

Mark A. Erle  
Electrical & Computer Engr. Dept.  
Lehigh University  
Bethlehem, PA 18015, USA  
mae5@lehigh.edu

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 two novel designs for fixed-point decimal multiplication that utilize decimal carry-save addition to reduce the critical path delay. First, a multiplier that stores a reduced number of multiplicand multiples and uses decimal carry-save addition in the iterative portion of the design is presented. Then, a second multiplier design is proposed with several notable improvements including fast generation of multiplicand multiples that do not need to be stored, the use of decimal (4:2) compressors, and a simplified decimal carry-propagate addition to produce the final product. When multiplying two  $n$ -digit operands to produce a  $2n$ -digit product, the improved multiplier design has a worst-case latency of  $n + 4$  cycles and an initiation interval of  $n + 1$  cycles. Three data-dependent optimizations, which help reduce the multipliers' average latency, are also described. The multipliers presented can be extended to support decimal floating-point multiplication.*

## 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 two-state computers [1]. Additional issues favoring binary arithmetic include better error characteristics [2, 3] and less hardware to implement the same function [4]. 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 [5], and humanity's natural\* affinity for decimal arithmetic [6]. In fact, due to the importance of decimal arithmetic in commercial applications, specifications for it are being added to the revised version of the IEEE Standard for Floating-Point Arithmetic [7]. These specifications are expected to be more comprehensive than the IEEE Standard for Radix-Independent Floating-Point Arithmetic [8], 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 [9], a dedicated decimal floating-point hardware implementation is likely to be

---

\*Human beings have ten fingers.

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 [10] and the IEEE Standard for Floating-Point Arithmetic, currently under revision [7].

Over the years, several designs for fixed-point decimal multiplication have been proposed, including [11, 12, 13, 14]. As these are patents or technical disclosure bulletins, implementation details are limited. These designs iterate over the digits of the multiplier and, based on the value of the current digit, either successively add the multiplicand or add a multiple of the multiplicand generated via costly lookup tables. Assuming decimal carry-propagate addition can complete in a single cycle, the designs employing an iterative addition approach require an average of  $5 \cdot n_1$  cycles, where  $n_1$  is the number of significant digits in the smaller of the two operands. There is a recent publication [15] that describes the algorithm and hardware design for the fixed-point decimal multiplier on the IBM z900 processor. The goal of that design is to implement decimal multiplication in the fewest cycles with the available hardware, which includes a 15-digit fixed-point decimal adder and a register file used for storing multiples of the multiplicand. That design requires an average of  $11 + 2.4 \cdot n_1$  cycles when the result is 15 digits or less and  $11 + 4.4 \cdot n_1$  cycles when the result is more than 15 digits. The iterative portion of the algorithm has a 15-digit decimal carry-propagate adder on the critical path.

The decimal multipliers presented in this paper use dedicated hardware to operate at high frequencies with relatively low latencies. These multipliers extend previous techniques used for decimal multiplication including generating a reduced set of multiplicand multiples [16], using carry-save addition for the iterative portion of the multiplier [11, 12], and using direct decimal addition [17] to implement decimal carry-save addition. Novel features of the second multiplier design presented in this paper include the use of decimal (3:2) counters and decimal (4:2) compressors, fast generation of multiplicand multiples that do not need to be stored, and a simplified decimal carry-propagate addition to produce the final product.

The outline of the paper is as follows. In Section 2, background information on decimal addition and multiplication is presented. The benefits and implications of multiplication via decimal carry-save addition and an initial decimal multiplier design are described in Section 3. A second decimal multiplier design with several notable improvements is described in Section 4. Then, in Section 5, three data-dependent optimizations that can be applied to the improved design are described. Section 6 contains a summary of the paper.

## 2: Background of Decimal Addition and Multiplication

Since decimal addition is a fundamental operation in the design of decimal multipliers, this section first gives an overview of techniques for decimal addition, followed by techniques for decimal multiplication. It is assumed that operands are in BCD format. 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. For example,

$P_i$  corresponds to the partial product after  $i$  iterations.

## 2.1: Decimal Addition

In a 4-bit BCD digit, six of the sixteen possible bit combinations are unused (i.e., ‘1010’ to ‘1111’). Because of this, BCD addition is more complicated than binary addition. When the sum of two BCD digits surpasses the maximum value of nine, the unused combinations should be skipped, the sum wrapped around, and the carry-out set to one. To accomplish this, one can bias a BCD digit with six, perform the binary addition of this biased digit and a second BCD digit, and then remove the bias if a carry is not produced.

Another approach is to perform *direct decimal addition*; namely, implementing logic that accepts as inputs two 4-bit BCD digits,  $x_i$  and  $y_i$ , along with a 1-bit carry-in,  $c_i[0]$ , and directly produces a 4-bit BCD sum digit,  $s_i$ , and a 1-bit carry-out,  $c_{i+1}[0]$ , such that

$$(c_{i+1}[0], s_i) = x_i + y_i + c_i[0] \quad (1)$$

where the weight of  $c_{i+1}[0]$  is 10 times the weight of  $s_i$ . The reader is referred to [17] for a detailed description of this approach. The equations for performing the direct decimal addition of two BCD digits are given below, where bit 0 is the LSB of the 4-bit BCD digit.

$$\begin{aligned} g_i[j] &= x_i[j] \cdot y_i[j] \quad 0 \leq j \leq 3 \quad \text{“generate”} \\ p_i[j] &= x_i[j] + y_i[j] \quad 0 \leq j \leq 3 \quad \text{“propagate”} \\ h_i[j] &= x_i[j] \oplus y_i[j] \quad 0 \leq j \leq 3 \quad \text{“half-adder”} \\ \\ k_i &= g_i[3] + (p_i[3] \cdot p_i[2]) + (p_i[3] \cdot p_i[1]) + (g_i[2] \cdot p_i[1]) \\ l_i &= p_i[3] + g_i[2] + (p_i[2] \cdot g_i[1]) \\ c_i[1] &= g_i[0] + (p_i[0] \cdot c_i[0]) \quad \text{“carry out of 1’s position”} \\ \\ s_i[0] &= h_i[0] \oplus c_i[0] \\ s_i[1] &= ((h_i[1] \oplus k_i) \cdot \overline{c_i[1]}) + ((\overline{h_i[1] \oplus l_i}) \cdot c_i[1]) \\ s_i[2] &= (\overline{p_i[2] \cdot g_i[1]}) + (\overline{p_i[3] \cdot h_i[2] \cdot p_i[1]}) + ((g_i[3] + (h_i[2] \cdot h_i[1])) \cdot \overline{c_i[1]}) + \\ &\quad ((\overline{p_i[3] \cdot p_i[2] \cdot p_i[1]}) + (g_i[2] \cdot g_i[1]) + (p_i[3] \cdot p_i[2])) \cdot c_i[1] \\ s_i[3] &= ((\overline{k_i \cdot l_i}) \cdot \overline{c_i[1]}) + (((g_i[3] \cdot \overline{h_i[3]}) + (\overline{h_i[3] \cdot h_i[2] \cdot h_i[1]})) \cdot c_i[1]) \\ c_{i+1}[0] &= k_i + (l_i \cdot c_i[1]) \end{aligned}$$

The above equations describe a functional block that can be used to implement either decimal carry-propagate addition or decimal carry-save addition, described in Section 3. When used to perform decimal carry-save addition, this block is called a decimal (3:2) counter.

## 2.2: Decimal Multiplication

Decimal multiplication performs the computation  $P = A \cdot B$ , where  $A$  is the multiplicand,  $B$  is the multiplier, and  $P$  is the product. It is assumed that  $A$  and  $B$  are each  $n$  digits and  $P$  is maximally  $2n$  digits.

A straightforward approach to decimal multiplication is to iterate over the digits of the multiplier,  $B$ , and based on the value of the current digit,  $b_i$ , successively add multiples of  $A$  to a product register [16]. The multiplier is typically traversed from least significant digit to most significant digit, and the product register is shifted one digit to the right after each iteration, which corresponds to division by 10. This approach allows an  $n$ -digit adder to be used to add the multiples of  $A$  to the partial product register and one product digit to be retired each iteration. The multiples  $2A$  through  $9A$  can be calculated at the start of the algorithm and stored along with  $A$  to reduce delay. The equation for this iterative approach to decimal multiplication is as follows:

$$P_{i+1} = (P_i + A \cdot b_i) \cdot 10^{-1} \quad (2)$$

where  $P_0 = 0$  and  $0 \leq i \leq n - 1$ . After  $n$  iterations,  $P_n$  corresponds to the final product  $P$ . The most notable shortcomings of this approach are the significant area or delay to generate the eight multiples, and the eight additional registers needed to store the multiples.

An alternative to storing all the multiples (called *primary multiples* or a *primary set*) is storing a reduced set of multiples. For example, if  $2A$ ,  $3A$ ,  $4A$ , and  $8A$  are precalculated and stored along with  $A$ , all the other multiples can be obtained dynamically with, at most, a single addition. This reduced set of multiples is called a *secondary set*, as no more than two members of the set need to be added to generate a missing multiple. Another reduced set of multiples is  $A$ ,  $2A$ ,  $4A$ , and  $8A$  [12]. Advantages of this set are one fewer multiple and a one-to-one correspondence with the weighted bits of a BCD digit. A disadvantage is the increase in delay or area needed to handle a *tertiary set* of multiples (i.e. the dynamic creation of  $7A$  requires the addition of three multiples:  $A$ ,  $2A$ , and  $4A$ ).

If secondary multiples are used, Equation 2 is replaced by the following equation to describe the iterative portion of a decimal multiplier:

$$P_{i+1} = (P_i + A \cdot b'_i + A \cdot b''_i) \cdot 10^{-1} \quad (3)$$

In Equation 3,  $A \cdot b'_i$  and  $A \cdot b''_i$  are secondary multiples which together equal the proper primary multiple (i.e.,  $A \cdot b'_i + A \cdot b''_i = A \cdot b_i$ ). Although the secondary multiple approach reduces the delay or area and register count, it introduces the overhead of potentially one more addition in each iteration. The next section describes a multiplier design that uses decimal carry-save addition to reduce this overhead.

### 3: Initial Multiplier Design

It is the intention of this research to design a decimal multiplication algorithm suitable for high-performance microprocessors with short cycle times. Since the multiplier may need to handle operands up to 34 decimal digits in length [18], it is unlikely a single decimal carry-propagate addition, let alone the two additions shown in Equation 3, can be performed in one cycle.

A substantial improvement in delay can be obtained by using decimal (3:2) counters (see Section 2.1 for equations). Using the decimal carry-save addition approach, Equation 3 can be expressed in two equations:

$$(PS'_i, PC'_i) = PS_i + PC_i + A \cdot b'_i \quad (4)$$

$$(PS_{i+1}, PC_{i+1}) = (PS'_i + PC'_i + A \cdot b''_i) \cdot 10^{-1} \quad (5)$$

where  $PS_i$  is the partial product sum comprised of 4-bit BCD digits and  $PC_i$  is the partial product carry (1 bit for each  $PS$  digit) at the end of the  $i^{th}$  iteration. Equations 4 and 5 form the basis of the initial multiplier design.

Figure 1 is a diagram of a decimal multiplier that implements Equations 4 and 5 in a single cycle. The top portion of the design, which is above the partial product register, performs the iterative equations. The bottom portion generates and stores the multiples of  $A$ , and produces the final product. To better understand the design, note that the same decimal carry-propagate adder is used to generate the multiples at the beginning of the multiplication and produce the final product at the end. As shown, the algorithm takes four cycles to generate the secondary multiples ( $2A$ ,  $3A$ ,  $4A$ , and  $8A$ ), one cycle for each multiplier digit to add up all the multiples, and one cycle to produce the final product. Thus, it has a latency of  $n + 5$  cycles and can begin a new multiplication every  $n + 5$  cycles.

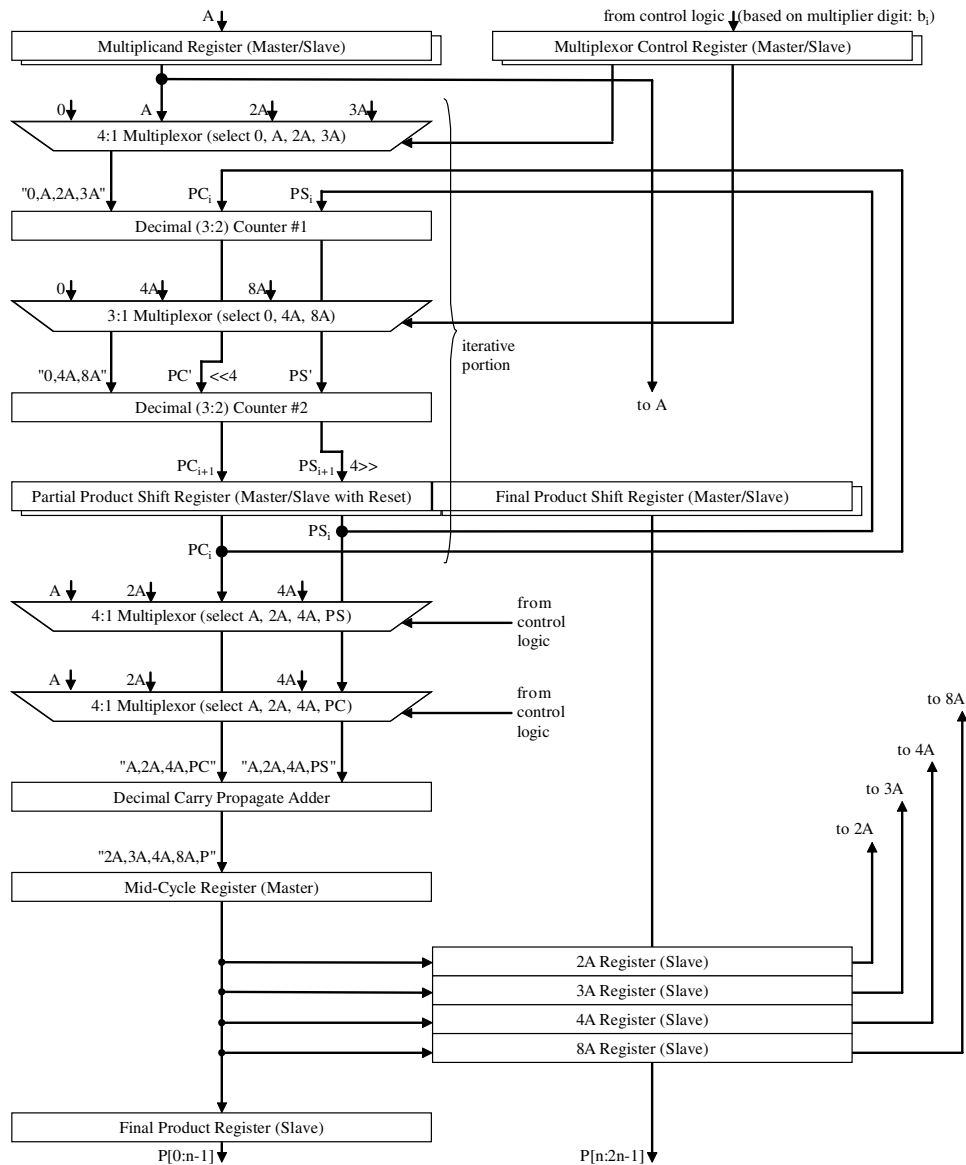
Both figures for the multiplier designs contain registers that are identified as “Master/Slave”, “Master”, or “Slave”. This is because the designs are implemented with a two-phase clock. The latches in the “Master” registers are active when the clock is high (phase 1), and the latches in the “Slave” registers are active when the clock is low (phase 2). Also noteworthy is the absence of selection logic on the data inputs to the registers. (For example, in Figure 1, the output of the “Mid-Cycle Register” toward the bottom of the design is fed to five different “Slave” registers.) The selecting of the data is accomplished through the use of gated clocks. In addition to reducing the delay in the dataflow portions of the design, this strategy also reduces the power consumption.

The design of Figure 1 has two frequency limiting issues. First, the two decimal (3:2) counters in series, along with the multiplexors needed to steer the appropriate multiples into the counters, contain too much logic for a single fast cycle. Unfortunately, simply splitting the logic into two pipeline stages is undesirable as it doubles the number of cycles needed for the iterative portion of the algorithm. Second, the decimal carry-propagate adder used to generate the multiples and the final product is too slow as well. Splitting this logic into two pipes is reasonable in this case as the overall latency only increases by three cycles. This is because the four secondary multiples can be produced in six cycles (two additional) and the final product produced in two cycles (one additional).

#### 4: Improved Multiplier Design

Reducing the delay of the iterative additions without significantly increasing the overall latency will yield the greatest performance improvement. This can be accomplished by reordering the addends in Equations 4 and 5, such that the secondary multiples are first added in one cycle using carry-save addition. Then in the following cycle, the sum and carry vectors from this addition are added to the partial product sum and carry and right shifted to produce the new partial product. Since the addition of the secondary multiples does not depend on the previous partial product, this reorganization allows the decimal carry-save additions to be split into two stages and operate at a much higher frequency. Using this approach, Equations 4 and 5 can be rewritten as:

$$\begin{aligned} (TS_i, TC_i) &= A \cdot b'_i + A \cdot b''_i & (6) \\ (PS_{i+1}, PC_{i+1}) &= (PS_i + PC_i + TS_i + TC_i) \cdot 10^{-1} & (7) \end{aligned}$$



**Figure 1. Initial Multiplier Design**

where  $TS_i$  and  $TC_i$  comprise the multiple of  $A$  in sum and carry format that is to be added based on  $b_i$ . Only the first pass through the decimal (3:2) counter in Equation 6 contributes to the overall latency since the partial product is never fed back through this counter. When implementing Equation 6, a simplified decimal (3:2) counter can be used, since only two decimal digits are added together ( $c_i[0] = 0$ ).

One impact of this improvement is that each digit summation in Equation 7 can have a maximum value of 20, as each sum digit can have a maximum value of nine and there are two carry bits. A decimal (4:2) compressor, analogous to a binary (4:2) compressor, is used to handle this increased complexity. A decimal (4:2) compressor accepts as inputs

two 4-bit BCD digits,  $x_i$  and  $y_i$ , and two 1-bit carry-ins,  $c_i[0]$  and  $c'_i[0]$ , and produces a 4-bit BCD sum digit,  $s_i$ , and a 1-bit carry-out  $c_{i+1}[0]$ . The decimal (4:2) compressor uses a standard decimal (3:2) counter to compute

$$(c''_{i+1}[0], s'_i) = x_i + y_i + c_i[0] \quad (8)$$

where  $c''_{i+1}[0]$  is an intermediate carry-out and  $s'_i$  is an intermediate sum. A simplified decimal (3:2) counter is then used to compute the final sum and carry as:

$$(c_{i+1}[0], s_i) = s'_i + c'_i[0] + c''_i[0] \quad (9)$$

As mentioned, the carry-propagate adder of Figure 1 used to generate the multiples and the final product is too slow for a high-frequency design. If  $2A$ ,  $4A$ , and  $5A$  are chosen as the secondary multiple set, the carry-propagate adder is only needed to produce the final product. These multiples can be generated quickly, since with both doubling and quintupling of BCD numbers there is no carry propagation beyond the next more significant digit [16]. When any BCD digit is doubled, its LSB initially becomes zero. Thus, when a carry-out of one occurs (for digit values in the range 5 – 9), it does not propagate beyond the next more significant digit's LSB, which is always zero. The equations for generating  $2A$  are as follows (bit 0 is the LSB):

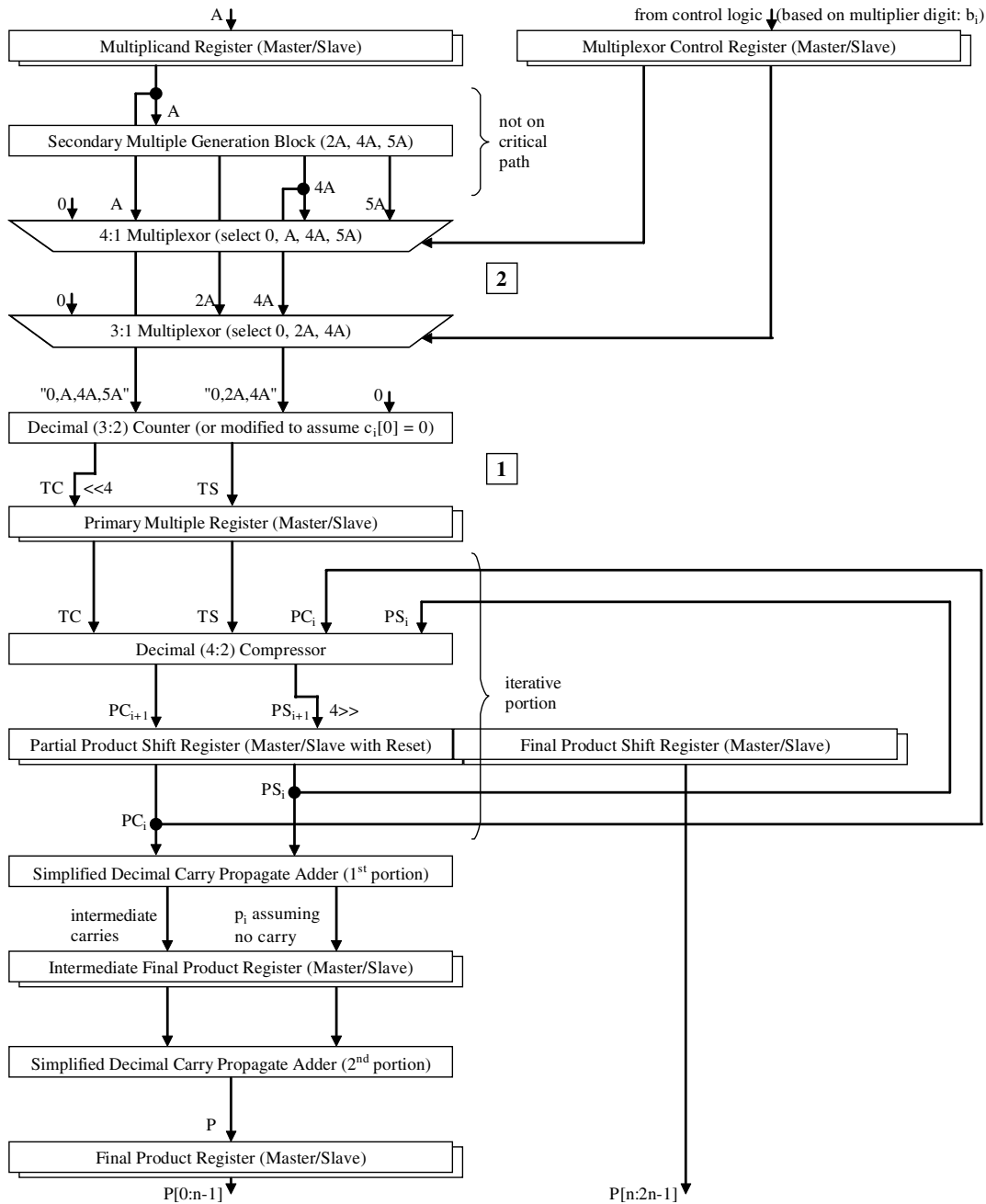
$$\begin{aligned} x2_i[0] &= (a_{i-1}[2] \cdot a_{i-1}[1] \cdot \overline{a_{i-1}[0]}) + (a_{i-1}[2] \cdot a_{i-1}[0]) + a_{i-1}[3] \\ x2_i[1] &= (\overline{a_i[3]} \cdot \overline{a_i[2]} \cdot a_i[0]) + (a_i[2] \cdot a_i[1] \cdot \overline{a_i[0]}) + (a_i[3] \cdot \overline{a_i[0]}) \\ x2_i[2] &= (a_i[1] \cdot a_i[0]) + (\overline{a_i[2]} \cdot a_i[1]) + (a_i[3] \cdot \overline{a_i[0]}) \\ x2_i[3] &= (a_i[2] \cdot \overline{a_i[1]} \cdot \overline{a_i[0]}) + (a_i[3] \cdot a_i[0]) \end{aligned}$$

For quintupling, note that any digit whose value is odd will initially become a five ('0101'), and any digit whose value is even will initially become a zero. Further, any digit whose value is  $\geq 2$  will produce a carry-out in the range 1 – 4. Thus, when a carry-out does occur, it will not propagate beyond the next more significant digit, which has a maximum value of five before adding the carry. The equations for generating  $5A$  are as follows (bit 0 is the LSB):

$$\begin{aligned} x5_i[0] &= (a_i[0] \cdot \overline{a_{i-1}[3]} \cdot \overline{a_{i-1}[1]}) + (\overline{a_i[0]} \cdot a_{i-1}[1]) + (a_i[0] \cdot a_{i-1}[3]) \\ x5_i[1] &= (\overline{a_i[0]} \cdot a_{i-1}[2]) + (a_i[0] \cdot \overline{a_{i-1}[2]} \cdot a_{i-1}[1]) + (a_{i-1}[2] \cdot \overline{a_{i-1}[1]}) \\ x5_i[2] &= (a_i[2] \cdot \overline{a_{i-1}[3]} \cdot \overline{a_{i-1}[1]}) + (a_i[0] \cdot \overline{a_{i-1}[2]} \cdot a_{i-1}[1]) + (\overline{a_i[0]} \cdot a_{i-1}[3]) \\ x5_i[3] &= (a_i[0] \cdot a_{i-1}[2] \cdot a_{i-1}[1]) + (a_i[0] \cdot a_{i-1}[3]) \end{aligned}$$

Consequently, in three gate delays,  $2A$  can be generated from  $A$ , in three more gate delays  $4A$  can be generated from  $2A$ , and in parallel  $5A$  can be generated from  $A$ . Thus, after six gate delays, which can easily fit into one cycle, all the multiples are ready. Further, since each of the multiples is generated from  $A$  using dedicated logic and the value of  $A$  does not change throughout the iterations, it is not necessary to store any of the multiples of  $A$  in registers. Referring to Equation 6, a 4-to-1 multiplexor selects  $A \cdot b'_i$  and a 3-to-1 multiplexor selects  $A \cdot b''_i$  based on the value of the current multiplier digit. The controls for these multiplexors are generated and latched one cycle before they are needed.

Another advantage of the decimal carry-save addition approach to decimal multiplication is the redundant form of the partial product, since each partial-product digit is a 4-bit BCD



**Figure 2. Improved Multiplier Design**

sum and a 1-bit carry. This means that the adder is less complex and faster than one which must accommodate two 4-bit sums in each digit position.

The improved design, which includes the changes just described, is shown in Figure 2. The multiplier takes one cycle to generate the secondary multiples ( $A$ ,  $2A$ ,  $4A$ , and  $5A$ ),

one cycle to produce the first multiple to be added, one cycle for each multiplier digit to add up all the multiples, and two cycles to produce the final product. Thus, it has a worst-case latency of  $n + 4$  cycles and an initiation interval of  $n + 1$  cycles. This design scales well to larger operand sizes, since increasing the operand size only affects the number of iterations and the delay of the final carry-propagate adder, which can be further pipelined.

## 5: Data-dependent Optimizations

There are a number of data-dependent optimizations that can be applied to the described multipliers. These include: skipping zeros, double-digit multiplication, and early exit. The early exit strategy is the only optimization of the three that does not add delay to the dataflow portion of the design. Although the skipping zeros and double-digit multiplication will add delay, the delay is not introduced in the critical iterative portion of the design.

### 5.1: Skipping Zeros

Assuming the value of every digit in the multiplier is equally likely, zeros will be encountered, and can be skipped, 10% of the time. The skipping can be implemented as follows. The control logic described in the aforementioned designs examines the next more significant digit the cycle before the dataflow adds its respective multiple. If this digit is a 0, then the control logic can skip this digit and examine the next more significant digit and proceed as before.

Even though a multiple does not need to be added when a particular multiplier digit is zero, the increasing order of each successive multiplier digit must be accounted for. Thus, the multiple needs to be left shifted one digit with respect to the partial product (Alternatively, the partial product can be shifted but this would add delay to the critical path). The shift can take the form of a two-input multiplexor on the output of the decimal counter (see [1](#) in Figure 2). The costs of this optimization are 1) the introduction of a single-digit shift operation in the dataflow, 2) the widening of the latch boundary between the decimal counter and the decimal compressor to store one more sum digit and carry bit, 3) the widening of the decimal compressor to accommodate one additional sum digit and carry bit, and 4) the generation and distribution of a skip signal to control the shifting.

### 5.2: Double-Digit Multiplication

Double-digit multiplication is an extension of an algorithm described in [13] and the skipping zeros concept. Instead of the control logic successively examining a single multiplier digit at a time, it examines two. If the necessary multiples can enter the first decimal counter through different input ports, then both can be added simultaneously (see [2](#) in Figure 2). For the design described in Section 4, up to 20% of each two digit grouping can be accomplished in a single iteration. These combinations are: 00, 01, 02, 04, 05, 10, 12, 14, 20, 21, 24, 25, 40, 41, 42, 44, 45, 50, 52, and 54.

The order of the more significant digit in the two digit grouping must be accounted for. Thus, one of the selected secondary multiples needs to be left shifted one digit with respect to the other multiple. This shift can take the form of a two-input multiplexor on each input of the decimal counter. The costs of this optimization are 1) the introduction of a single-digit shift operation that must take place in the dataflow (on each input of the

decimal counter), 2) the widening of the decimal counter to accommodate one additional digit, 3) the widening of the latch boundary between the decimal counter and the decimal compressor to store one more digit and carry bit, 4) the widening of the decimal compressor to accommodate one additional digit and carry, and 5) the generation and distribution of two skip signals to control the shifting.

If the cycle time permits, each multiplexor that selects among a subset of the secondary multiples could be widened to allow selection among all of them. This would raise the potential for double-digit multiplication to 25% by enabling the following additional combinations: 11, 15, 22, 51, and 55.

### 5.3: Early Exit

An early exit from the multiplication can occur after the most significant non-zero digit in the multiplier has been accounted for [14]. The multiplier operand can be examined to find the leading, non-zero digit while the secondary multiples are being calculated. Once this digit is found, the next more significant digit can be replaced with '1100'. As this is an invalid BCD digit, the control logic can treat this digit as the end of the multiplier operand and exit early. The benefit of this optimization can be significant, albeit difficult to quantify. The costs are 1) the introduction of hardware to detect the leading, non-zero digit in the multiplier operand and to replace the next more significant digit with '1100', and 2) the modification of the control logic to detect a digit whose value is '11xx' and to exit early.

To further improve the benefit of this optimization, the operand with the most leading zeros could be used for the multiplier, as is done in [15]. The costs are 1) the introduction of hardware to compare the number of leading zeros in  $A$  and  $B$ , and 2) the introduction of multiplexors to conditionally swap the values of  $A$  and  $B$ . A drawback of this strategy is that it increases the overall latency since the first step in the algorithm, the generation of the secondary multiples, cannot begin until the multiplier operand is chosen and latched.

## 6: Summary

A justification for decimal arithmetic hardware and a motivation for decimal multiplication, in particular, were presented. To introduce the concepts and issues of decimal multiplication, and to establish a baseline for comparison, an initial design of a decimal multiplier was shown. Then, the shortcomings of the initial design were examined and alternative solutions were presented. A second design of a decimal multiplier with several notable improvements was shown to have a significantly higher operating frequency, a reduced worst-case latency of  $n + 4$  cycles, and an initiation interval of  $n + 1$  cycles, where  $n$  is the number of significant digits in the multiplier operand. Finally, several data-dependent optimizations were described along with estimates of their usefulness in reducing the average latency.

## Acknowledgments

This work is sponsored in part by International Business Machines.

## References

- [1] W. S. Brown and P. L. Richman, "The Choice of Base," *Comm. of the ACM*, vol. 12, pp. 560–561, Oct 1969.
- [2] R. P. Brent, "On the Precision Attainable with Various Floating-Point Number Systems," *IEEE Trans. Comp.*, vol. C, pp. 601–607, Jan 1973.
- [3] R. W. Hamming, "On the Distribution of Numbers," *Bell Syst. Tech. J.*, vol. 49, pp. 1609–1625, Oct 1970.
- [4] W. Buchholz, "Fingers or fists? (The Choice of Decimal or Binary Representation)," *Communications of the ACM*, vol. 2, no. 12, pp. 3–11, 1959.
- [5] A. Tsang and M. Olschanowsky, "A Study of Database 2 Customer Queries," IBM Technical Report 03.413, IBM, San Jose, CA, Apr 1991.
- [6] 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.
- [7] IEEE Standards Committee, "IEEE Standard for Floating-Point Arithmetic." <http://754r.ucbtest.org/drafts/754r.pdf>, February 2003.
- [8] Technical Committee on Microprocessors and Microcomputers of the IEEE Computer Society, "IEEE Standard for Radix-Independent Floating-Point Arithmetic," ANSI/IEEE Std 854-1987, IEEE, New York, NY, Oct 1987.
- [9] M. A. Erle, J. M. Linebarger, and M. J. Schulte, "Potential Speedup Using Decimal Floating-Point Hardware." Submitted to the 36th Asilomar Conference on Signals, Systems and Computers, Nov 2002.
- [10] M. Cowlshaw, "General Decimal Arithmetic Specification." <http://www2.hursley.ibm.com/decimal/decarith.html>, January 2003.
- [11] R. H. Larson, "High Speed Multiply Using Four Input Carry Save Adder," *IBM Technical Disclosure Bulletin*, pp. 2053–2054, December 1973.
- [12] T. Ohtsuki, Y. Oshima, S. Ishikawa, K. Yabe, and M. Fukuta, "Apparatus for Decimal Multiplication," *U.S. Patent*, Jun 1987. #4,677,583.
- [13] R. L. Hoffman and T. L. Schardt, "Packed Decimal Multiply Algorithm," *IBM Technical Disclosure Bulletin*, vol. 18, pp. 1562–1563, October 1975.
- [14] J. J. Bradley, B. L. Stoffers, T. R. S. Jr., and M. A. Widen, "Simplified Decimal Multiplication by Stripping Leading Zeros," *U.S. Patent*, Jun 1986. #4,615,016.
- [15] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough, "The IBM z900 Decimal Arithmetic Unit," in *Conference Record of the Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1335–1339, Nov 2001.
- [16] R. K. Richards, *Arithmetic Operations in Digital Computers*. New Jersey: D. Van Nostrand Company, Inc., 1955.
- [17] M. S. Schmookler and A. W. Weinberger, "High Speed Decimal Addition," *IEEE Trans. Computers*, vol. C-20, pp. 862–867, Aug 1971.
- [18] M. Cowlshaw, "Decimal Arithmetic Specification Encoding." <http://www2.hursley.ibm.com/decimal/decbits.html>, February 2003.