# Decimal Floating-Point Multiplication Via Carry-Save Addition

Mark A. Erle
International Business Machines
6677 Sauterne Drive
Macungie, PA 18062
merle@us.ibm.com

Michael J. Schulte and Brian J. Hickmann
University of Wisconsin – Madison
Dept. of Electrical and Computer Engineering
Madison, WI 53706
schulte@engr.wisc.edu and bjhickmann@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 the design of a decimal floating-point multiplier that complies with specifications for decimal multiplication given in the draft revision of the IEEE 754 Standard for Floating-point Arithmetic (IEEE 754R). This multiplier extends a previously published decimal fixed-point multiplier design by adding several features including exponent generation, sticky bit generation, shifting of the intermediate product, rounding, and exception detection and handling. The core of the decimal multiplication algorithm is an iterative scheme of partial product accumulation employing decimal carry-save addition to reduce the critical path delay. Novel features of the proposed multiplier include support for decimal floating-point numbers, on-the-fly generation of the sticky bit, early estimation of the shift amount, and efficient decimal rounding. Area and delay estimates are provided for a verified Verilog register transfer level model of the multiplier.*

## 1. Introduction

Due to the importance of decimal arithmetic in commercial applications and the potential speedup achievable [7], microprocessors which support decimal floating-point arithmetic will soon be available [19]. Further, specifications for decimal arithmetic have been added to the revised version of the IEEE Standard for Floating-Point Arithmetic [12] (hereafter referred to as "IEEE 754R"). These specifications are more comprehensive than those specified in the radix-independent standard [8], including formats and operations for single, double, and quadruple precision Decimal Floating-Point (DFP) numbers.

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 presents the design of a decimal floating-point multiplier in compliance IEEE 754R and a prevailing decimal arithmetic specification [3].

Over the years, several designs for fixed-point decimal multiplication have been proposed, including [5, 9, 13, 14]. These designs iterate over the digits of the multiplier and, based on the value of the current digit, either successively add the multiplicand or a multiple of the multiplicand. The multiples are generated via costly lookup tables or developed using a subset of previously generated multiples. None of these designs support floating-point arithmetic. Only a few previous papers present designs for decimal floating-point multiplication [1, 2]. The multiplier designs by Cohen et al. [2] and Bolenger et al. [1] are digit-serial and have long latencies. Furthermore, the results they produce do not comply with IEEE 754R.

The decimal multiplier presented in this paper extends a previously published fixed-point decimal multiplier [5]. That design features a reduced set of multiplicand multiples [16], the use of carry-save addition for the iterative portion of the multiplier [13, 14], and the use of direct decimal addition [18] to implement decimal carry-save addition. Novel features of the proposed multiplier include support for DFP arithmetic, on-the-fly generation of the sticky bit, early estimation of the shift amount, and efficient decimal rounding, which does not exhibit rounding overflow.

Throughout this paper, upper case variables denote multiple digit words, lower case variables with subscripts denote 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 $i$. Square brackets are not needed when the lower-case variable represents a binary number. 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 digit in the intermediate product word after $i$ iterations. Super-

scripts are used to differentiate various forms of the same variable. Bits and digits are indexed from most significant to least significant, starting with index zero. A subscript next to a constant or string indicates the base. As for terminology, the *precision of a number* is the maximum number of digits able to be represented in a given fixed-width format, the *number of significant digits* is the number of digits from the most significant non-zero digit to the least significant digit (LSD), inclusive, and the *number of essential digits* is the number of digits between the most significant non-zero digit and the least significant non-zero digit, inclusive.

The outline of the paper is as follows. In Section 2, background information on decimal floating-point multiplication is presented. This section includes information on IEEE 754R's storage formats and the concept of a preferred exponent. Next, Section 3 contains a flowchart of the algorithm along with descriptions of the major components of the proposed design, including exponent processing, intermediate product shifting, and rounding. An implementation of the proposed DFP multiplier design is described in Section 4 including area and delay numbers from synthesis. Section 5 contains a summary of the paper.

## 2. Background of DFP Multiplication

A DFP number may be expressed in the following form:

$$D = -1^s \cdot C \cdot 10^{E-bias} \tag{1}$$

where $s$ is the sign bit, $C$ is the non-negative integer significand, and $E$ is the biased non-negative integer exponent. The significand is an integer, which differs from normalized Binary Floating-Point (BFP) significands (e.g., $1.01\ldots$). To maintain consistency between DFP and BFP formats, IEEE 754R defines the parameters for various format widths for both DFP and BFP numbers in a slightly modified scientific notation form. The modification is the radix point appearing to the right of the Most Significant Digit (MSD) or Most Significant Bit (MSB), respectively, thereby restricting the magnitude of the significands to less than their radix. The primary parameter affected by this decision is that of the exponent. When the significand is viewed in this restricted-magnitude form, IEEE 754R uses the variable $e$ for the exponent; where $emin \leq e \leq emax$. When the significand is viewed in integer form, IEEE 754R uses the variable $q$ (*quanta*) for the exponent; where $emin - (p-1) \leq q \leq emax - (p-1)$ and $p$ is the precision. The $(p-1)$ term reconciles the alternative exponents since the radix points for both significand representations with the same precision are separated by $p-1$ digits. $E$ relates to IEEE 754R's $e$ and $q$ in the following way: $E = q+bias = e-(p-1)+bias$. In the spirit of making DFP similar to elementary arithmetic, this paper uses the integer form of the significand.

The non-normalized significand leads to a few distinct differences between DFP and binary floating-point (BFP) arithmetic. First, when aligning significands such that digits of the same order are located in the same physical position for add-type operations, both operands may need to be shifted. Second, for multiply operations, if the number of significant digits in the unrounded product exceeds the format's precision, $p$, then this intermediate product may need to be left shifted prior to rounding. Third, if an intermediate product contains $p - i$ essential digits, then there exists $i$ equivalent representations of the value. Note the $i$ possible representations can only be realized if there is sufficient available exponent range to allow the leading non-zero digit to be placed in the MSD position and the trailing non-zero digit to be placed in the LSD position. For example, if $p$ equals 5 and the operation is $32 \times 10^{15}$ multiplied by $70 \times 10^{15}$, then possible results are $22400 \times 10^{29}$, $2240 \times 10^{30}$, or $224 \times 10^{31}$ (leading zeros not shown).

Because of the possibility of multiple representations of the same value, IEEE 754R introduces the concept of a preferred exponent. The preferred exponent, $PE$, drawn from elementary arithmetic, is based on the operation and the exponent(s) of the operand(s). For multiplication, the preferred exponent, prior to any rounding or exceptions, is:

$$PE = E^A + E^B - bias \tag{2}$$

For example, the product of $A = 320 \times 10^{-2}, (E^A = -2+101)$ multiplied by $B = 70 \times 10^{-2}, (E^B = -2+101)$ is $P = 22400 \times 10^{-4}, (PE = -4+101)$. If an intermediate product with leading zeros and raised to the preferred exponent has essential digits to the right of the decimal point, the significand is left shifted while decrementing the intermediate exponent to yield a product with the maximum number of significant digits (so long as the exponent stays in range).

Although IEEE 754R specifies multiple storage formats of DFP numbers and two encondings for the significand within each format, the multiplier presented in this paper assumes the operands are stored in the *decimal64* format with Densely Packed Decimal (DPD) encoding. The *decimal64* format is comprised of a 1-bit sign, a 13-bit combination field, and a 50-bit trailing significand field, which after decoding yields a 10-bit binary exponent and a 16-digit decimal significand. The choice of precision, exponent base and range, and significand representation and encoding is examined in [4], and to a lesser extent in [12].

## 3. DFP Multiply Algorithm and Components

The floating-point multiplier design presented in this paper is an extension of a fixed-point multiplier design published in [5]. A flowchart-style drawing of the decimal floating-point multiplication algorithm is shown in Figure 1,

with the steps of the decimal fixed-point multiplier from [5] surrounded by a gray box. The solid arrows indicate the general flow of execution, and the dashed arrows indicate a transfer of information. The operation begins with the read-

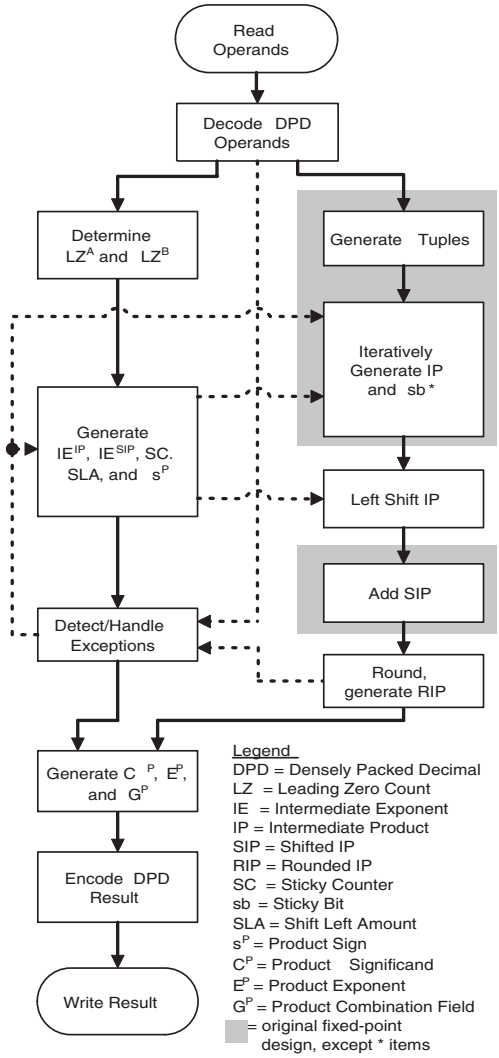


**Figure 1. Flowchart of Decimal Multiplier**

ing of the operands from either a register file or from memory. As the operands are encoded via the DPD algorithm, each must be decoded. Next, the double, quadruple, and quintuple of the multiplicand are generated in the datapath portion of the design. Then, in an iterative manner starting with the LSD of the multiplier significand, each digit is used to select two multiples to add together to yield a respective partial product, the partial product is added with the previous iteration's accumulated product, and the accumulated product is shifted one digit to the right. After all the multiplier digits have been processed, the sum of all the partial products, called the intermediate product, is stored in a register twice as long as each input significand. All the additions in the iterative portion of the algorithm yield intermediate results in decimal carry-save form. For a diagram of the top portion of the decimal floating-point multiplier, the reader is referred to [6]. The top portion of the design, with the exception of the sticky bit generation logic, is the decimal fixed-point multiplier described in [5].

In parallel with the generation of the multiples and the accumulation of partial products, the significands are examined to determine their leading zero counts, $LZ^A$ and $LZ^B$, the exponents are examined to determine the various intermediate exponents, $IE^{IP}$ and $IE^{SIP}$, and the signs are XORed to determine the product sign, $s^P$. Based on the leading zero counts and the intermediate exponent, two vital control values are generated: a shift left amount, $SLA$, and a sticky counter, $SC$. The shift left amount is needed to properly align the intermediate product, $IP$, prior to rounding. The sticky counter is needed to generate the sticky bit, $sb$, created on-the-fly during the accumulation of partial products. The intermediate product is then shifted based on the shift left amount to produce the shifted intermediate product, $SIP$. Using the operands' combination fields, the intermediate exponent, and information from the shift and round steps, a determination is made as to whether an exception needs to be signaled and corrective action taken.

At the end of the iterative accumulation of partial products, the intermediate product is in the $2p$-digit intermediate product register. Ultimately, a $p$-digit rounded product needs to be delivered. Since the shifted intermediate product is in carry-save form, an add step is necessary to produce a non-redundant product. After rounding the shifted intermediate product, the product exponent, $E^P$, product significand, $C^P$, and product combination field are generated*. Finally, these values, along with the product sign, are used to produce a final result which is DPD encoded and written to a register file or memory.

A critical design choice is the location of the decimal point in the datapath as this dictates the direction the intermediate product may need to be shifted and the location and implementation of the rounding logic. For this design, the location is chosen to be exactly in the middle of the intermediate product register. This keeps the decimal point in the same location throughout the datapath. Further, the intermediate product need only be shifted in one direction to produce a rounded product (except when underflow occurs).

In the remainder of this section, the primary components necessary to perform decimal floating-point multiplication are described in detail. These include generating the intermediate exponent, shifting the intermediate product, generating the sticky bit, generating the result sign, rounding, and

*In actuality, the leading digit of $C^P$ and the leading two bits of $E^P$ are contained in the combination field.

detecting and handling any exceptions.

### 3.1. Intermediate Exponent Generation

At the end of partial product accumulation, $p$ digits of the intermediate product are to the right of the decimal point. Thus, the intermediate exponent of the intermediate product, $IE^{IP}$, is the preferred exponent increased by $p$:

$$\begin{aligned} IE^{IP} &= E^A + E^B - bias + p \\ &= PE + p \end{aligned} \quad (3)$$

After left shifting the intermediate product as part of preparing a $p$-digit final product, the intermediate exponent is decreased by the shift left amount, $SLA$ (described in the next subsection). The exponent associated with this shifted intermediate exponent, $IE^{SIP}$, is calculated as follows.

$$\begin{aligned} IE^{SIP} &= E^A + E^B - bias + p - SLA \\ &= PE + p - SLA \\ &= IE^{IP} - SLA \end{aligned} \quad (4)$$

The shifted intermediate product is then rounded, and the associated exponent is named the intermediate exponent of the rounded intermediate product, $IE^{RIP}$. The $IE^{RIP}$ is one less than $IE^{SIP}$ or equal to $IE^{SIP}$ depending on whether or not a corrective left shift of one digit occurs during rounding. However, the product exponent may differ from $IE^{RIP}$ due to an exception.

Related to the intermediate exponent calculations is the determination of the amounts by which $IE^{IP}$ is less than the minimum exponent, $Emin$, or more than the maximum exponent, $Emax$. These comparisons are used to increase or decrease the shifting of the intermediate product in an effort to prevent an exception. As the shifting of the intermediate product may be affected by these comparisons, the generation of the sticky bit must be altered accordingly. The computation of the shift amount for the intermediate product and the generation of the sticky bit are described in the next two subsections, and the handling of extreme numbers is described in Subsections 3.4 and 3.5.

### 3.2. Intermediate Product Shifting

As mentioned earlier, the intermediate product may need to be shifted to achieve the preferred exponent or to bring the product exponent into range. Calculating the shift amount is dependent upon, among other things, the number of leading zeros in the intermediate product. However, instead of waiting until the intermediate product is generated to count the leading zeros, the latency of the multiplication is reduced by determining a shift amount based on the leading zeros in both the multiplicand and multiplier significands. With this approach, the pre-calculated shift amount may be off by one since the number of significant digits in the final product may be one less than the sum of the significant digits of each significand. Thus, the product may need to be left shifted one additional digit at some point after the initial shift.

Except when $IE^{IP} < Emin$, the shift is always to the left[†]. This is because each partial product is added to the previous accumulated product with its LSD one digit to the right of the decimal point. With an estimate of the significance of the intermediate product based on the significance of each significand, $S^{IP} = S^A + S^B$, the shift left amount is determined as follows. If $S^{IP} > p$, then one or more leading zeros of the intermediate product may need to be shifted off to the left to maximize the significance of the result. However, if $S^{IP} \leq p$, then the entire product will reside solely in the lower half of the intermediate product register (assuming all the multiplier significand digits have been processed). In the latter case, the less significant half of the intermediate product register can be placed into the upper half, by left shifting the intermediate product $p$ digits. These two situations lead to the following equation for the shift left amount, $SLA$.

$$\begin{aligned} SLA &= min((2 * p) - (S^A + S^B), p) \\ &= min((2 * p) - ((p - LZ^A) + (p - LZ^B)), p) \\ &= min(LZ^A + LZ^B, p) \end{aligned} \quad (5)$$

where $LZ^A$ and $LZ^B$ are leading zero counts of the significands, $C^A$ and $C^B$, respectively.

In the event the actual significance of the intermediate product is one digit less than the estimated significance, it may be necessary to left shift the intermediate product one more digit after the initial left shift. The potential for a corrective left shift of one digit necessitates maintaining an additional digit to the right of the decimal point. This digit is referred to as the *guard* digit and is analogous to the guard bit used in binary floating-point multiplication. The handling of the final left shift by one digit occurs in the rounding portion of the algorithm and is described in Subsection 3.5.

The shift left amount, as estimated above, is dependent on all the multiplier significand's digits being processed. One design option is to exit the iterative portion of the algorithm after processing the most significant nonzero digit of the multiplier significand. Although this option complicates the processor's instruction issue and completion logic, substantial cycles may be saved for certain workloads. If early exit is to be supported, the shift left amount calculation, as well as other design components, needs to be altered accordingly.

[†]This assumes every multiplier digit is processed during the partial product accumulation portion of the multiplication algorithm.

### 3.3. Sticky Determination

After left shifting, any nonzero digits in the less significant half of the intermediate product register must be evaluated in the context of the rounding mode to determine if rounding up is necessary. As mentioned in the previous subsection, a corrective left shift of one digit may be needed if the actual significance of the intermediate product is one less than the sum of the significands' significance. In the event the corrective left shift is performed and the guard digit is shifted into the LSD position of the more significant half of the intermediate product register, the next digit must be maintained such that it can be determined if the remaining digits are less than one half the Unit in the Last Place (ULP), exactly one half ULP, or greater than one half ULP. Since this digit in the next less significant position to the guard digit is critical to rounding, it is called the *round* digit, which is analogous to the round bit in binary multiplication. The bits of the digits to the right of the round digit can all be logically ORed to produce a *sticky* bit.

To improve the latency and area of floating-point multiplication, it is desirable to know *a priori* which digits will be used in the sticky bit calculation. Having such knowledge allows the sticky bit to be generated on-the-fly with less hardware and wiring resource than waiting until the entire intermediate product is available and then selecting which digits should be ORed together. This can be readily accomplished in this design as a non-redundant digit, formed during the accumulation of a new partial product, enters the MSD of the less significant half of the intermediate product register while the intermediate product is right shifted one position.

To determine when a digit being right shifted from the round digit position to the next less significant digit position should be included in the sticky bit generation, a counter is used. The starting value of this counter is initialized just prior to the first partial product entering the intermediate product register and cleared between operations. Whenever the counter value is greater than zero, the digit being shifted out of the round digit position is ORed with the previous sticky bit value, which is cleared between operations. The initial value of the sticky counter, $SC$, is generally the significance of the intermediate product minus the format's precision, unless this difference yields a negative number. Thus,

$$\begin{aligned} SC &= max(0, S^{IP} - p) \\ &= max(0, (p - LZ^A) + (p - LZ^B) - p) \\ &= max(0, p - (LZ^A + LZ^B)) \end{aligned} \quad (6)$$

Note the counter is decremented twice before any nonzero data enters the digit position to the right of the round digit position. This insures the two digits which end up in the guard and round digit positions after left shifting are not included in the sticky bit generation. Up to two more cycles can be taken to generate $SC$ so long as the value of the counter is correspondingly less than what is described in Equation 6. Also, if the intermediate product needs to be left shifted one additional digit, the sticky bit calculated in the manner above is still legitimate. This is because the guard digit will be moved into the LSD position of the product, and the round digit and sticky bit calculated in the manner above are all that is needed for rounding.

### 3.4. Sign Processing & Exception Handling

Sign processing is relatively straightforward. If the result is a number, the sign of the result is simply the XOR of the signs of the operands. However, if the result is NaN, IEEE 754R does not specify the value of the sign bit. For ease of implementation, the sign logic used when the result is a number is also used when the result is NaN.

There are four exceptions that may be signaled during multiplication: invalid operation, overflow, underflow, and inexact. The invalid operation exception is signaled when either operand is a signaling NaN or when zero and infinity are multiplied The default handling of the invalid operation exception involves signaling the exception and producing a quiet NaN for the result. If only one operand is a signaling NaN, then the quiet NaN result is created from the signaling NaN. Note that any produced or propagated NaN must have each ten-bit grouping in its trailing significand field as a defined pattern (i.e., only $1000/1024$ possible combinations are defined; see IEEE 754R). If one operand is a signaling NaN and the other is a quiet NaN, then this design converts the signaling NaN to a quiet NaN and returns this as the result[‡]. That is, the diagnostic information potentially contained in the signaling NaN is deemed more important than the diagnostic information potentially contained in the quiet NaN.

The overflow exception is signaled when a result's magnitude exceeds the largest finite number. The detection is accomplished after rounding by examining the computed result as though the exponent range is unlimited. Default overflow handling, as specified in IEEE 754R, involves the selection of either the largest normal number or canonical infinity and the raising of the inexact exception. Details on overflow handling appear in the next subsection. The inexact exception, an indication essential digits have been lost, is raised because the most significant nonzero digit in the shifted intermediate product has been shifted off the most significant end of the register, effectively, in an effort to decrease the exponent into range.

Under default exception handling, the underflow exception is signaled when a result is both tiny and inexact. Tini-

[‡]This is not required behavior in IEEE 754R.

ness is when the result's magnitude is between zero and the smallest normal number, exclusive. The detection of tininess can be accomplished in two ways, although each implementation of IEEE 754R must choose one way and use it for all operations. The first approach detects underflow prior to rounding by examining the computed result as though both the exponent range and the precision is unlimited. The second approach detects underflow after rounding by examining the computed result as though the exponent range is unlimited but with the result format's fixed precision. As an example, the smallest normal number in the *decimal32* format is $1000000 \times 10^{Emin-bias}$. If $7777000 \times 10^{Emin-bias}$ is multiplied by $9000000 \times 10^{bias-8}$, the intermediate product is $6999300.0000000 \times 10^{Emin-bias-1}$. However, this exponent cannot be represented. Instead of abruptly converting this number to zero, a subnormal number is produced by shifting the significand to the right one digit position and increasing the exponent by one to achieve the minimum exponent. Thus, the product significand is $0699930 \times 10^{Emin-bias}$. By reducing the precision in this manner, underflow occurs gradually. In the preceding example, the shifting to the right of the significand did not result in the loss of any nonzero digits. Thus, the results are exact, albeit subnormal, and the underflow exception is not raised. For an example of when the underflow exception is signaled, consider the following: $7777000 \times 10^{Emin-bias}$ multiplied by $9000000 \times 10^{bias-13}$. Here, the intermediate product is $6999300.0000000 \times 10^{Emin-bias-6}$. To achieve the minimum exponent, the significand must be right shifted and rounded[§] to produce $0000007 \times 10^{Emin-bias}$. Since one or more nonzero digits are "lost" to rounding, the result is both tiny and inexact and the underflow exception is signaled. Default underflow handling is explained in the next subsection.

### 3.5. Rounding

Rounding is required when all the essential digits of the intermediate product cannot be placed into the product coefficient or when overflow or underflow occurs. The description of each rounding mode required by IEEE 754R and its associated condition(s) are listed in Table 1. The default rounding mode is language-defined, but is encouraged to be round to nearest, ties to even.

In the case of rounding based solely on the number of essential digits, rounding is accomplished by selecting either the shifted intermediate product truncated to $p$ digits or its incremented value. In order to determine which value is to be selected, the following are needed: the rounding mode, the product's sign, the shifted intermediate product, including a guard digit, $g$, round digit, $r$, and sticky bit, $sb$, and an adder.

[§]The round to nearest, ties to even rounding mode is used.

The adder must be able to produce a non-redundant sum from its inputs, some of which may be in carry-save form. Further, it must be able to add a one into either its LSD position or its guard position. The two options for the position to inject a one are necessary as the estimate of the shift left amount may be off by one, in which case a corrective left shift is required. Though this may appear to require more than one adder, both situations can be supported by using a single compound adder. The inputs to the adder are the data in the $p$ MSD positions of the shifted intermediate product. To understand why it is sufficient to use only one compound adder $p$ digits wide, consider the four possibilities of adding a zero or a one into the LSD or guard digit positions. Clearly, adding a zero into the guard digit position is the same as adding a zero into the LSD position (so long as the original guard digit is concatenated). The remaining two possibilities are related in the following way. If a one is added into the guard digit position and a carry occurs out of the guard digit position (i.e., $g == 9$), then this is equivalent to adding a one into the LSD position and changing $g$ to zero. Conversely, if a carry does not occur out of the guard digit position (i.e., $g < 9$), then this is equivalent to adding a zero into the LSD position and concatenating the incremented guard digit.

Before presenting the rounding scheme employing the compound adder, the following simplification is offered. This design does not need to contend with rounding overflow. Rounding overflow is when the truncated intermediate product is incremented due to rounding and a carry out of the MSD position occurs. Rounding overflow cannot occur in this design due to the range of each operand and the manner in which the shift left amount is determined. To learn why rounding overflow cannot occur, the reader is referred to a proof in [6].

The use of a single compound adder to generate both a $p$-digit significand and its incremented value, and the guarantee of no post-rounding normalization, allows a simple and efficient rounding scheme to be developed which is unique from recent binary rounding schemes such as those presented and referenced in [15]. Here, $C^{+0}$ and $C^{+1}$ are used to represent the plus zero and plus one sums, respectively, emerging from the compound adder.

First, an indicator, $grsb$, is set whenever there are nonzero digits to the right of the LSD position of the shifted intermediate product. That is, $grsb = (g > 0) + (r > 0) + sb$. This indicator, when set, may lead to a corrective left shift if there is a leading zero in the compound adder's outputs. The corrective left shift does not happen when round up is to occur and the first $p$ digits of the shifted intermediate product are zero followed by all nines. In this case, a round up will produce a carry into the MSD position and the corrective left shift must be preempted. Fortunately, this unique case can be readily detected. It is the only situ-

**Table 1. Rounding Mode, Round Up Conditions, and Product Override for Overflow**

| Rounding Mode | Condition for Round-up – Non-overflow | Product Override – Overflow | |
|---|---|---|---|
| | | $s^{IP} == 0$ | $s^{IP} == 1$ |
| Nearest, ties to even | $(g > 5) + ((g == 5) \cdot (l[3] + (r > 0) + sb))$ | $+\infty$ | $-\infty$ |
| Nearest, ties away from zero | $g \geq 5$ | $+\infty$ | $-\infty$ |
| Toward positive infinity | $!s^P \cdot ((g > 0) + (r > 0) + sb)$ | $+\infty$ | largest finite $-$number |
| Toward negative infinity | $s^P \cdot ((g > 0) + (r > 0) + sb)$ | largest finite $+$number | $-\infty$ |
| Toward zero (truncate) | *none* | largest finite $+$number | largest finite $-$number |

ation in which the MSD of $C^{+0}$ is a zero and the MSD of $C^{+1}$ is a nonzero.

Next, for the given rounding mode, two round up values, $ru^{cls==0}$ and $ru^{cls==1}$, are computed for the cases of no corrective left shift by one and a corrective left shift by one, respectively. The computations are based on the shifted intermediate product and the round up condition(s) in Table 1. The difference between the two round up values is for the case of a corrective left shift, the guard digit is treated as the LSD, and the round digit is treated as the guard digit. As an example, if the rounding mode is round toward zero, both round up values are zero. As another example, if the rounding mode is round to nearest, ties away from zero and the LSD, guard, round, and sticky values are $0, 5, 0, 0$, then $ru^{cls==0} = 1$ and $ru^{cls==1} = 0$.

At this point, the guard digit must be incremented and an indicator developed when the original guard digit equals nine. The incremented guard digit is needed during a corrective left shift when round up is performed. The carry out of the decimal digit adder can be used as the $g == 9$ indicator, $g9$. However, it is important to note this carry out is never allowed to propagate into the LSD position of the compound adder.

Using $C^{+0}$, $C^{+1}$, $ru^{cls==0}$, $ru^{cls==1}$, $grsb$, $g$, $g + 1$, and $g9$, the algorithm presented in Figure 2 realizes rounding for the decimal floating-point multiplier design. The algorithm is presented as three distinct cases involving the MSDs of the two conditional sums, $C[0]^{+0}$ and $C[0]^{+1}$. The fourth case, when the plus zero sum has a zero in its MSD and the plus one sum has a nonzero digit in its MSD, cannot happen.

Though the rounding scheme of Figure 2 may appear complex, the choice is simply between $C^{+0}$, $C^{+1}$, or these values left shifted one digit with either $g$ or $g + 1$ concatenated. For those cases in which a left-shifted form of a conditional sum is chosen, the intermediate exponent of the shifted intermediate product is decremented.

In the case of rounding due to overflow, the product is rounded according to column IV in Table 1. The table describes the product to be generated for each rounding mode under default exception handling as specified in IEEE 754R.

Ultimately, the detection of overflow occurs by comparing the intermediate exponent of the rounded intermediate product, $IE^{RIP}$, with the maximum exponent, $Emax$. However, the following steps are taken prior to this comparison in an effort to keep the intermediate exponent in range. If the intermediate exponent of the intermediate product ($IE^{IP}$ in Equation 3) minus the shift left amount ($SLA$ in Equation 5) is greater than $Emax$, then $SLA$ is increased to subsequently decrease the intermediate exponent of the shifted intermediate product ($IE^{SIP}$ in Equation 4). $SLA$ can only be increased to the extent all the leading zeros of the intermediate product are removed. If there are only enough leading zeros to bring $IE^{SIP}$ down to at least $Emax+1$, then it is possible there will be one more leading zero in the shifted intermediate product than estimated, and a left shift of one digit can occur during rounding to prevent overflow. Note the sticky counter ($SC$ in Equation 6) must be decreased by the same amount $SLA$ is increased. After adjusting $SLA$ and $SC$, shifting the intermediate product, and rounding the shifted intermediate product, $IE^{RIP}$ is compared to $Emax$. If $IE^{RIP} > Emax$, then overflow has occurred and the rounding mode and product sign are used to select a product based on Table 1. Both the overflow and inexact exceptions are signaled.

Averting underflow is similar to that described for overflow. The calculated $SLA$ must be decreased by the amount which would lead $IE^{SIP}$ to drop below $Emin$. The sticky counter must be increased by the same amount $SLA$ is decreased. Dissimilar from overflow, however, is the following behavior. If $IE^{IP} < Emin$, then $SLA$ is set to zero, and the intermediate product is shifted to the right to bring the $IE^{SIP}$ into range. To accomplish the right shift, the iterative portion of the algorithm is allowed to continue beyond the processing of all the multiplier significand digits. Partial products of value zero are used in the additional iterations so as not to alter the value of the intermediate product. The number of additional iterations is equal to the amount by which $Emin$ exceeds $IE^{IP}$. Note the sticky counter must be increased by the number of additional iterations. The number of additional iterations need not exceed $p + 2$ as this amount is guaranteed to place the most significant nonzero digit of any accumulated product beyond the round digit position. Thus, with a properly adjusted sticky counter, all the intermediate product data is ORed into the sticky bit. After clearing $SLA$, adjusting

"No leading zeros, no corrective left shift"

1. $C[0]^{+0} != 0$ and $C[0]^{+1} != 0$
   (a) $ru^{cls==0} == 0 \Rightarrow C^P = C^{+0}$
   (b) $ru^{cls==0} == 1 \Rightarrow C^P = C^{+1}$

"Leading zeros, possible corrective left shift"

2. $C[0]^{+0} == 0$ and $C[0]^{+1} == 0$
   (a) $grsb == 0$
      i. $IE^{SIP} == PE$ or $IE^{SIP} \leq Emin$
      $\Rightarrow C^P = C^{+0}$
      ii. $IE^{SIP} > PE$ and $IE^{SIP} > Emin$
      $\Rightarrow C^P = (C^{+0} \ll 1) \,||\, g$
   (b) $grsb == 1$ and $IE^{SIP} \leq Emin$
      i. $ru^{cls==0} == 0 \Rightarrow C^P = C^{+0}$
      ii. $ru^{cls==0} == 1 \Rightarrow C^P = C^{+1}$
   (c) $grsb == 1$, $IE^{SIP} > Emin$, and $ru^{cls==1} == 0$
   $\Rightarrow C^P = (C^{+0} \ll 1) \,||\, g$
   (d) $grsb == 1$, $IE^{SIP} > Emin$, and $ru^{cls==1} == 1$
      i. $g9 == 0 \Rightarrow C^P = (C^{+0} \ll 1) \,||\, (g+1)$
      ii. $g9 == 1 \Rightarrow C^P = (C^{+1} \ll 1) \,||\, (g+1)$,
      note $g + 1 == 0$

"Zero followed by all nines"

3. $C[0]^{+0} == 0$ and $C[0]^{+1} != 0$
   (a) – (c) same as in Case 2
   (d) $grsb == 1$, $IE^{SIP} > Emin$, and $ru^{cls==1} == 1$
   $\Rightarrow C^P = C^{+1}$

**Figure 2. Rounding Scheme**

$SC$, "shifting" the intermediate product, and rounding the shifted intermediate product, $IE^{RIP}$ is compared to $Emin$. If $IE^{RIP} < Emin$ and $grsb$ of the chosen compound adder output is one, then underflow has occurred. Additionally, if $IE^{RIP} == Emin$, the MSD of the chosen compound adder output is zero, and $grsb$ of the chosen compound adder output is one, then underflow has occurred. Both the underflow and inexact exceptions are signaled.

One design consideration is whether or not to allow underflow to have a variable latency. The number of additional iterations need only be $min(p+2, Emin - IE^{IP})$. However, to keep the processor's instruction issue and completion logic simple, it may be best to stall for $p+2$ cycles always. If handling underflow with fewer fixed cycles is desired, the existing left shifter can be altered to support right shifting as well. Since in this design the shifter is before the adder (i.e., the more significant half of the intermediate product is in carry-save form), the adder would need to be widened to support a greater number of digit positions containing carry-save data. The next section describes how the components in the previous subsections are combined to realize a decimal floating-point multiplier.

## 4. Decimal Floating-Point Multiplier Design

Figure 3 shows all the design components from the previous section together to realize DFP multiplication. The block-level drawing is of the bottom datapath portion of the decimal floating-point multiplier design, beginning with the $2p$-digit intermediate product register and a sticky bit that was generated on-the-fly (see Subsection 3.3). The top datapath portion of the design, ending with the same intermediate product register, is shown in [6]. Not shown in either design drawing is the control logic, which is where the intermediate exponents, the sticky counter, the shift left amount, and the rounding control are calculated. These calculations are not timing critical in a non-pipelined multiplier.

Referring again to Figure 3, the first step is to shift the intermediate product based on the shift left amount, $SLA$ (described in Subsection 3.2), and store the $p+2$ digit output in the shifted intermediate product register. The two additional digits are needed for the guard and round digits. Then, in support of the rounding scheme presented in Subsection 3.5, a compound adder receives the data stored in the shifted intermediate product register. Since the data are either in non-redundant form or in sum and carry form (four sum bits and one carry bit), a unique compound adder is needed. For the sum portion of the addition, the $sum_i + carry_i + 0$ and the $sum_i + carry_i + 1$ is generated for each digit position, $i$. For the carry portion of the addition, a digit generate equal to $s_i[0] \cdot s_i[3] \cdot c_i$ and digit propagate equal to $s_i[0] \cdot (s_i[3] + c_i)$ are produced. The digit propagate and generate are then fed into a carry network that generates the carries to select the appropriate digits to yield $C^{+0}$ and $C^{+1}$. The compound adder is only $p$ digits long, as described in Subsection 3.5. Once the two compound adder outputs are available in registers (i.e., $C^{+0}$ and $C^{+1}$), the rounding logic produces the product significand, $C^P$, based on the rounding scheme in Figure 2.

Notable implementation choices include leveraging the leading zero counts of the operands' signficands, passing NaNs through the dataflow with minimal overhead, and handling gradual underflow via minor modification to the control logic. Equations 5 and 6, and indirectly 4, use the leading zero counts of the significands. This is intentional as the determination of leading zeros is a common function in floating-point units [17]. Once each digit is identified as zero or nonzero, the generation of the leading zero count is the same as that for a BFP mantissa. As the accumulation of partial products is iterative, a single leading zero counter is used to determine successively the leading zero counts of $C^A$ and $C^B$. If an operand is NaN, that operand's NaN

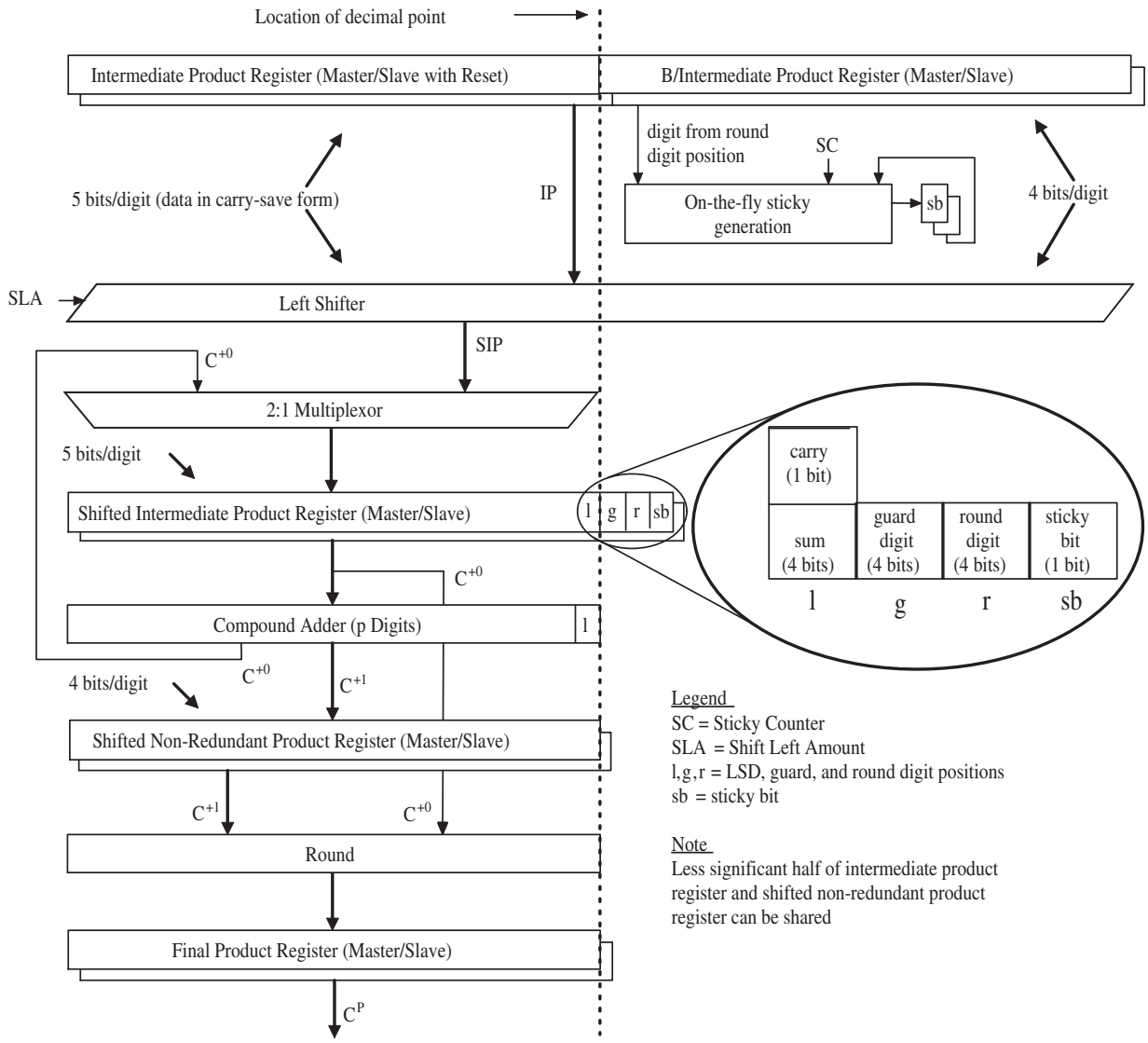


**Figure 3. Bottom Portion of Decimal Floating-Point Multiplier Design**

payload is used when forming the result. Instead of supporting alternative paths through the dataflow, the control logic passes $C^A$ through the dataflow by multiplying it by 1. If operand $B$ is NaN, $C^B$ is held in the less significant portion of the intermediate product register while the control logic overrides the shift left amount such that $C^B$ is left shifted into the more significant half of the shifted intermediate product register. As for gradual underflow, the control logic extends the iterative partial product accumulation portion of the algorithm and successively adds partial products equal to zero such that the accumulated partial product is right shifted until $IE^{IP}$ increases to $Emin$ or all nonzero data are shifted into the sticky bit. This support of gradual underflow extends the latency of the multiplier from 25 cycles to a maximum of $43 = 25 + (p + 2)$ cycles.

Register transfer level models of both the presented 64-bit DFP multiplier and its predecessor decimal fixed-point multiplier [5] were coded in Verilog. Both designs were synthesized using LSI Logic's gflxp 0.11um CMOS standard cell library and the Synopsys Design Compiler. To validate the correctness of the design, over $50,000$ testcases covering all rounding modes and exceptions were simulated successfully on the pre-synthesis design. Publicly available tests include directed-random testcases from IBM's FPgen tool [11] and directed tests soon to be available at [10]. Table 2 contains area and delay estimates for the DFP multiplier design presented and its predecessor decimal fixed-point multiplier design. The values in the *FO4 Delay* col-

umn are based on the delay of an inverter driving four same-size inverters having a $55ps$ delay, in the aforementioned technology. The critical path in the DFP multiplier is in the stage with the 128-bit barrel shifter, while for the decimal fixed-point multiplier it is the decimal $4:2$ compressor.

**Table 2. Area/Delay for Decimal Multipliers**

| Decimal Design | Cell Count | Area ($um^2$) | Delay ($ps$) | FO4 Delay |
|---|---|---|---|---|
| Floating-point | 117,627 | 237,607 | 850 | 15.45 |
| Fixed-point logic[a] | 32,347 | 65,341 | | |
| Data path logic[b] | 20,580 | 41,572 | | |
| Add/round | 16,489 | 33,308 | | |
| Left shift | 5,958 | 12,035 | | |
| Encode/decode | 1,768 | 3,571 | | |
| Control logic | 362 | 731 | | |
| Other[c] | 40,123 | 81,049 | | |
| Fixed-point [5] | 59,234 | 119,653 | 810 | 14.72 |

[a] Does not include the final adder.
[b] Includes $LZ^A$, $LZ^B$, $SC$, and $SLA$ generation, input registers, etc.
[c] Includes additional latches, interconnect, spacing, etc.

## 5. Summary

A justification for decimal arithmetic hardware and a motivation for decimal floating-point multiplication were presented. Next, the design components necessary to extend a previously published fixed-point decimal multiplier were described. The components include exponent generation, sticky bit generation, shifting of the intermediate product, rounding, and exception detection and handling. An algorithm and block-level drawing of a proposed DFP multiplier were presented. Novel features of the proposed multiplier include support for decimal floating-point numbers, on-the-fly generation of the sticky bit, early estimation of the shift amount, and efficient decimal rounding. All the presented design components, except the on-the-fly generation of the sticky bit, can be applied to pipelined floating-point decimal multipliers. Area and delay estimates were presented from the synthesized results of the verified register transfer level model.

## Acknowledgment

The authors thank Liang-Kai Wang for his contributions to the Verilog model of this design. This work is sponsored in part by International Business Machines.

## References

[1] G. Bohlender and T. Teufel. *Computer Arithmetic: Scientific Computation and Programming Languages*, chapter BAP-SC: A Decimal Floating-Point Processor for Optimal Arithmetic, pages 31–58. B. G. Teubner, Stuttgart, Germany, 1987.

[2] M. S. Cohen, T. E. Hull, and V. C. Hamacher. CADAC: A Controlled-Precision Decimal Arithmetic Unit. *IEEE Transactions on Computers*, C-32(4):370–377, April 1983.

[3] M. F. Cowlishaw. Decimal Floating-Point: Algorism for Computers. *16th Symposium on Computer Arithmetic*, pages 104–111, June 2003.

[4] M. F. Cowlishaw, E. M. Schwarz, R. M. Smith, and C. F. Webb. A Decimal Floating-Point Specification. In *15th Symposium on Computer Arithmetic*, pages 147–154. IEEE, IEEE Computer Society Press, July 2001.

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

[6] M. A. Erle, M. J. Schulte, and B. J. Hickmann. Decimal Floating-Point Multiplication Via Carry-Save Addition – Extended Version. World Wide Web, June 2007. http://mesa.ece.wisc.edu.

[7] 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*, volume 2, pages 1073–1077, November 2002.

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

[9] R. L. Hoffman and T. L. Schardt. Packed Decimal Multiply Algorithm. *IBM Technical Disclosure Bulletin*, 18(5):1562–1563, October 1975.

[10] IBM. General Decimal Arithmetic Testcases. World Wide Web. http://www2.hursley.ibm.com/decimal/dectest.html.

[11] IBM Floating-Point Test Generator. Floating-Point Test Suite for IEEE 754R Standard. World Wide Web. http://www.haifa.il.ibm.com/projects/verification/fpgen/ieeets.html.

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

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

[14] T. Ohtsuki, Y. Oshima, S. Ishikawa, K. Yabe, and M. Fukuta. Apparatus for Decimal Multiplication. *U.S. Patent*, June 1987. #4,677,583.

[15] N. T. Quach, N. Takagi, and M. J. Flynn. Systematic IEEE Rounding Method for High-Speed Floating-Point Multipliers. *IEEE Transactions on VLSI Systems*, 12(5):511–521, May 2004.

[16] R. K. Richards. *Arithmetic Operations in Digital Computers*. D. Van Nostrand Company, Inc., New Jersey, 1955.

[17] M. S. Schmookler and K. J. Nowka. Leading Zero Anticipation and Detection – A Comparison of Methods. In *15th Symposium on Computer Arithmetic*, page 7. IEEE, IEEE Computer Society Press, July 2001.

[18] M. S. Schmookler and A. W. Weinberger. High Speed Decimal Addition. *IEEE Transaction on Computers*, C(20):862–867, August 1971.

[19] S. Shankland. IBM's Power6 Gets Help with Math, Multimedia. World Wide Web, October 2006. http://news.zdnet.com/2100-9584_22-6124451.html.