A Parallel IEEE P754 Decimal Floating-Point Multiplier

Brian Hickmann, Andrew Krioukov, and Michael Schulte
University of Wisconsin - Madison
Dept. of Electrical and Computer Engineering
Madison, WI 53706
{bjhickmann, krioukov, and schulte}@wisc.edu

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

Abstract

Decimal floating-point multiplication is important in many commercial applications including banking, tax calculation, currency conversion, and other financial areas. This paper presents a fully parallel decimal floating-point multiplier compliant with the recent draft of the IEEE P754 Standard for Floating-point Arithmetic (IEEE P754). The novelty of the design is that it is the first parallel decimal floating-point multiplier offering low latency and high throughput. This design is based on a previously published parallel fixed-point decimal multiplier which uses alternate decimal digit encodings to reduce area and delay. The fixed-point design is extended to support floating-point multiplication by adding several components including exponent generation, rounding, shifting, and exception handling. Area and delay estimates are presented that show a significant latency and throughput improvement with a substantial increase in area as compared to the only published IEEE P754 compliant sequential floating-point multiplier. To the best of our knowledge, this is the first publication to present a fully parallel decimal floating-point multiplier that complies with IEEE P754.

1. Introduction

Decimal arithmetic is necessary in many financial and commercial applications, which process decimal values and perform decimal rounding. However, current software implementations are prohibitively slow [6], prompting hardware manufacturers such as IBM to add decimal floating-point(DFP) arithmetic support to upcoming microprocessors [18]. Furthermore, the IEEE 754 Working Group has recognized the importance of decimal arithmetic by adding it to the revised IEEE P754 Draft Standard for Floating-Point Arithmetic (IEEE P754) [11].

Previous decimal multipliers have primarily focused on fixed-point multiplication. Designs including [14, 8, 5, 12] use a sequential approach of iterating over the digits of the multiplier and selecting an appropriate multiple of the multiplicand. Generally, these designs have high latency and low throughput due to their sequential approach.

A few parallel fixed-point multiplier designs have also been proposed [1, 13]. The floating-point multiplier presented in this paper is based on the radix-10 fixed-point multiplier in [1] due to its highly efficient structure. This multiplier generates a sufficient subset of multiplicand multiples and then selects all the partial-products in parallel based on the digits of the multiplier operand. Only a few designs supporting DFP multiplication have been presented [3, 2, 15]. However, to our knowledge currently only the iterative multiplier from [15] complies with the IEEE P754 standard.

This paper presents a parallel DFP multiplier based on a parallel fixed-point multiplier [1] and a previous implementation of a DFP multiplier [15]. The novelty of the design is that it is the first parallel DFP multiplier, offering low latency, high throughput, and IEEE P754 compliance. In addition, novel early shift amount calculation and exception pass-through mechanisms are used to provide increased performance. This design allows trade-offs between clock frequency and overall latency by adding pipeline stages. As compared to the sequential design in [15], an 11-stage pipelined version of our design has similar clock speed, significantly reduced latency (11 vs. 21 cycles), and one result per cycle throughput, while incurring a substantial 371% increase in area. To the best of our knowledge, this is the first published design of a parallel decimal floating-point multiplier that is compliant with IEEE P754.

The outline of the paper is as follows. In Section 2, background information on decimal floating-point multiplication and IEEE P754 formats is presented. Section 3 contains a detailed description of the design; starting with a high-level overview, followed by descriptions of the fixed-point multiplier, intermediate exponent and shift calculation, rounding, and special case handling. Results are presented in Section 4, followed by a summary in Section 5.

2. Background

The IEEE P754 standard specifies formats for both binary floating-point (BFP) and decimal floating-point (DFP) numbers [11]. The primary difference between the two formats, besides the radix, is the normalization of the significands. BFP significands are normalized with the radix point to the right of the most significant bit (MSB), while DFP significands are not required to be normalized and are typically represented as integers. In this paper, all DFP operands use integer significands.

The IEEE P754 standard specifies DFP formats of 32, 64, and 128 bits. An IEEE P754 DFP number contains a sign bit, an integer significand with a precision of *p* digits, and a biased exponent. The value of a finite DFP number is:

$$D = -1^{s} \times C \times 10^{E-bias} \quad (1)$$

where s is the sign bit, C is the non-negative integer significand, and E is the biased non-negative integer exponent.[1] The significand can be encoded either in binary or in Densely Packed Decimal (DPD) [4], which in the draft standard is referred to as the decimal encoding. The exponent must be in the range $[Emin, Emax]$, when biased by $bias$. Representations for infinity and Not-a-Number (NaN) are also provided.

The non-normalized significand allows redundant representations of numbers. For example, multiplying 32×10^{15} by 70×10^{15} yields a result that could be represented as 22400×10^{29}, 2240×10^{30}, or 224×10^{31}. Because of the possibility of multiple representations, IEEE P754 defines a preferred exponent, which for multiplication is:

$$PE = E^{A} + E^{B} - bias \quad (2)$$

where E^{A} and E^{B} are the biased exponents of the first and second operands, respectively. The multiplier uses the preferred exponent when encoding the result of a multiplication, so long as this does not lead to a loss of precision. In the above example, this implies that $2240 * 10^{30}$ is chosen as the result. However, in the case of multiplying $9812345678912345 \times 10^{10}$ by 2×10^{5} with a precision of $p = 16$ digits, the result of $1962469135782469 \times 10^{16}$ is chosen to ensure there is no loss of precision.

The multiplier design presented in this paper uses 64-bit DFP numbers with significands encoded in the DPD format. This format has $p = 16$ decimal digits of precision in the significand, an unbiased exponent range of $[-383, 384]$, and a $bias$ of 398.

[1] Biased exponents in this paper represented by E relate to IEEE P754's exponents by: $E = e - (p-1) + bias$ where e is the unbiased exponent as defined in the IEEE P754 Draft Standard.

3. Multiplier Design

A general overview of our parallel floating-point multiplier design is presented in Figure 1. Arrows are used to indicate the direction of data flow and dashed lines separate the main stages of the design.

The multiplication begins with reading two operands in IEEE P754 format and decoding each to produce the sign bit, significand, exponent, and flags for special values of Not-a-Number (NaN) or infinity. The significands of the two operands are then decoded from the DPD encoding to Binary Coded Decimal (BCD).

As soon as the decoded significands become available, a decimal fixed-point multiplication begins. If one or both of the operands is a NaN, its value is preserved through the multiplier by forcing the other operand to a value of one. The fixed-point multiplier generates 16 decimal partial products in parallel and adds them along with a possible correction term using a carry-save adder (CSA) tree followed by a high-speed decimal carry-propagate adder. The result is a non-redundant, 32-digit BCD number referred to as the intermediate product (IP). Details on the fixed-point multiplier are provided in Subsection 3.1 and special case handling is discussed in Subsection 3.5.

In parallel with the multiplication, a shift-left amount (SLA) and corresponding intermediate exponent of the intermediate product (IE^{IP}) are calculated for shifting the 32-digit fixed-point result to fit into $p = 16$ digits of precision. This calculation is performed using leading-zero detection (LZD) on both operands in order to estimate the number of significant digits in the result. Since the calculation occurs prior to the computation of the IP, the SLA may be off by one due to the significance of the product being one less than expected. In addition to the SLA and IE^{IP} values, this unit calculates the sign bit of the final result and detects exception conditions which are later used by the rounding unit. A detailed description of the SLA and IE^{IP} generation unit is provided in Subsection 3.2.

The 32-digit IP from the multiplier is then shifted by the SLA amount, forming the shifted intermediate product (SIP). The design sets the decimal point to be in the middle of the 32-digit intermediate product, thus splitting it into a 16-digit truncated product (TP^{+0}) and a 16-digit fractional product (FRP). This design choice keeps the decimal point in the same location throughout the datapath and requires only a left-shift to produce the SIP.

Next, the FRP is used to produce the guard digit, round digit, and sticky bit. In parallel, the TP^{+0} is incremented to allow the rounding logic to select between TP^{+0} and TP^{+1}, which is sufficient to support all rounding modes.

Finally, the rounding and exception logic uses the rounding mode and exception conditions to select between TP^{+0}, TP^{+1}, and special case values to produce the rounded inter-

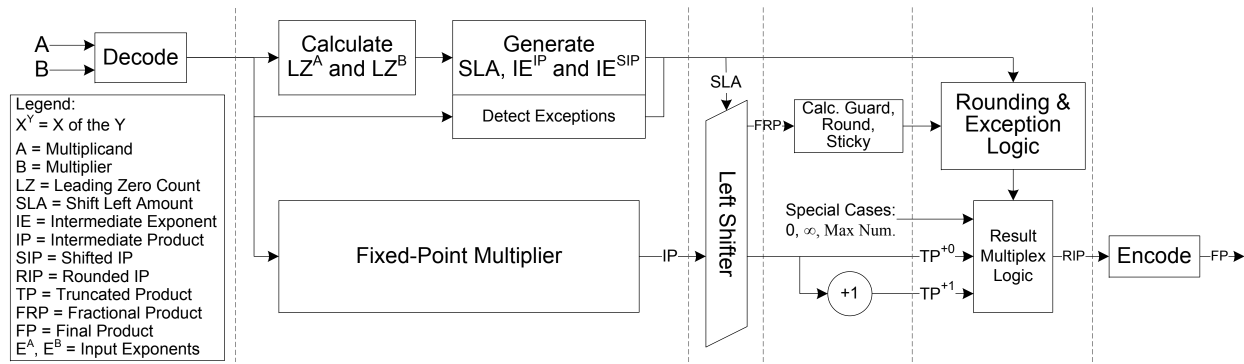

Figure 1. High-level Decimal Floating-Point Multiplier Diagram

mediate product (RIP). The RIP is then encoded in DPD and put in IEEE P754 format with the appropriate flags set to produce the final product (FP). Rounding and exceptions are covered in detail in Subsections 3.4 and 3.5.

To illustrate the multiplication process, consider multiplying $3141592654 \times 10^{-9}$ by 988121822×10^{11} when the rounding mode is roundTiesToEven, as depicted in Figure 2. Our parallel multiplier starts by decoding the inputs from IEEE P754 format. Next, fixed-point multiplication is performed on the integer significands. In parallel, the SLA value of 13 is calculated by summing the leading zero counts of both operands and the unbiased version of the IE^{IP} value is calculated as $11 + (-9) + p = 18$. After the fixed-point multiplication is complete, the result (IP) is shifted left by $SLA = 13$ digits to form the shifted intermediate product (SIP). The truncated integer (TP) portion of the result is then incremented, and rounding is performed to select between TP^{+0} and TP^{+1} as the final product (FP). More details on the multiplier design are given in the following subsections.

3.1. Fixed Point Multiplier

A key component of the design is the fixed-point multiplier, which is based on the radix-10 parallel fixed-point multiplier presented in [1]. The novelty of this fixed-point design is in using a special BCD digit recoding to reduce logic and in generating partial products in parallel. The fixed point multiplier unit takes two 16-digit integer operands, calculates 16 partial products in parallel and returns their sum, a 32-digit integer.

There are three main components in the fixed-point multiplier design: generation of multiplicand multiples, selection of partial products, and reduction of partial products. The multiple generation stage produces $\{1x...5x\}$ multiples of the multiplicand. Next, positive or negative versions of the multiplicand multiples are selected based on the Booth sign-encoded digits of the multiplier operand. All of the selected multiples (partial products) are then reduced using a carry-save adder (CSA) tree followed by a high-speed decimal carry-propagate adder.

In the first stage of the process, $\{1x...5x\}$ multiples of the multiplicand are generated. All of the resulting products are encoded in BCD-4221 to simplify the CSA tree.[2] The $1x$, $2x$, $4x$ and $5x$ multiples are generated in a novel way using digit recoding and wired shifts, however, the $3x$ multiple requires a BCD adder to sum $1x$ and $2x$.

The next stage uses the digits of the multiplier to select, in parallel, 16 multiplicand multiples which form the partial products. Signed-digit recoding is used to require only the $\{1x...5x\}$ multiples and their complements. If a multiplier digit is greater than 5, a negative multiple is selected and $1x$ is added to the next digit (e.g. $7 = -3 + 10$). In this way every multiplier digit selects $\pm\{1x...5x\}$. As a special case, if the most significant digit (MSD) of the multiplier is greater than 5, then an extra corrective partial product of $1x$ is added to the partial product tree.

After the partial products have been selected they are reduced using a CSA tree. Since all 16 combinations of the BCD-4221 encoded partial-products are valid decimal values, a slightly modified binary CSA tree is used. The tree layout is based on the tree presented in [1], but is modified to add a final corrective partial product to support multiplier operands with MSDs > 5. The resulting carry and partial sum use the BCD-8421 encoding and are added using a high-speed direct decimal carry-propagate adder [17] to produce a 32-digit BCD-8421 result. The direct decimal adder uses a Kogge-Stone network to quickly produce the carries between digits.

[2] In this paper, we represent alternate decimal encodings in the format BCD-xxxx where the x's are the weights of each binary bit. For example, 1001_2 has a value of $4 + 0 + 0 + 1 = 5$ with the BCD-4221 encoding and a value of $8 + 0 + 0 + 1 = 9$ with the BCD-8421 encoding.

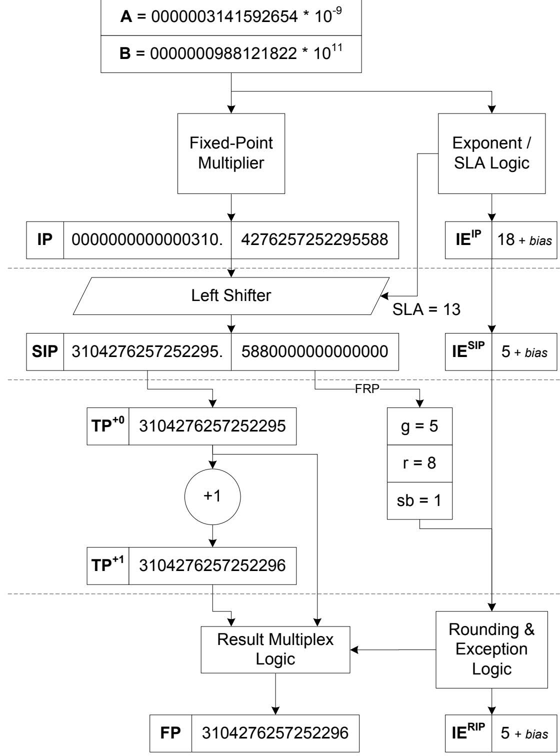

Figure 2. DFP Multiply Example

3.2. Intermediate Exponent and Shift Calculation

Two key computations that occur in parallel with the fixed-point multiplication are exponent calculation and shift left amount calculation. At the end of the fixed point multiplication, *p* digits of the intermediate product are to the right of the decimal point. This increases the intermediate exponent of the intermediate product (IE^{IP}) by *p*, giving the following equation for the biased intermediate exponent:

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

Since there are significant digits to the right of the decimal point, the intermediate product produced by the fixed point multiplier may need to be shifted to achieve the preferred exponent or to bring the product exponent into range. Except in the subnormal case when $IE^{IP} < Emin$, this shift is always to the left due to the placement of the decimal point. The shift amount depends primarily on the number of leading zeros in the intermediate product. Normally, this requires the multiplier to wait until the intermediate product is generated before counting the leading zeros. To reduce the latency of the multiplication, the shift amount is precalculated using the leading zero counts of both the multiplicand and the multiplier significands. This approach introduces a possible one-digit error in the shift amount, since the number of significant digits in the intermediate product may be one less than the sum of the significant digits of the operands. Thus, the product may need to be left shifted by one more digit in the result multiplexing logic.

The shift left amount is determined by first calculating the estimated number of significant digits in the intermediate product, S^{IP}, where S^{IP} is the sum of the number of significant digits of the two operands as given by $S^{IP} = S^A + S^B$. Next, two cases are considered. If $S^{IP} > p$ then some number of zeros need to be shifted off to the left in order maximize the significance of the product. If $S^{IP} \leq p$, then all of the significant digits of the product are to the right of the decimal point, and the shift value should be *p*, in order to place the result to the left of the decimal point. This leads to the following equations for the shift left amount or SLA.

$$\begin{aligned} SLA &= min((2p) - (S^A + S^B), p) \\ &= min((2p) - ((p - LZ^A) + (p - LZ^B)), p) \\ &= min(LZ^A + LZ^B, p) \end{aligned} \quad (4)$$

After the left shift is performed, the intermediate exponent is decreased by the *SLA*. This intermediate exponent associated with the shifted intermediate product is calculated simply as $IE^{SIP} = IE^{IP} - SLA$.

As mentioned earlier, the intermediate product may require a corrective left shift by one digit within the result multiplexing logic. This requires that an extra digit be retained to the right of the decimal point to handle this case. This digit is referred to as the guard digit, *g*. If a corrective left shift is performed, then the intermediate exponent for the rounded intermediate product is $IE^{RIP} = IE^{SIP} - 1$, otherwise it is $IE^{RIP} = IE^{SIP}$.

In order to prevent an unnecessary overflow or underflow exception or when the generation of subnormal numbers is detected, the above equations for the intermediate exponent and shift left amount are modified to attempt to keep the exponent within range. These modifications are described along with other exception conditions in Subsections 3.3 and 3.5.

3.3. Overflow and Underflow

Efficiently handling overflow and underflow in decimal multiplication provides a significant challenge and requires changes to SLA generation, exponent calculation, and rounding. For overflow, detection is done after rounding by comparing the intermediate exponent of the rounded intermediate product (IE^{RIP}) with the maximum exponent, *Emax*. Before rounding, however, several steps are taken to avoid overflow. During exponent and *SLA* generation, if $IE^{IP} - SLA > Emax$, as given in Equations 3 and

4, then our design attempts to avoid overflow by increasing *SLA* in order to decrease IE^{SIP}. The *SLA* value can only be increased to the point where all leading zeros of the intermediate product have been removed. That is, the following must hold: $SLA_{new} \leq (LZ^A + LZ^B)$. If the maximum SLA yields $IE^{SIP} = Emax + 1$, then it is still possible to avoid overflow if a corrective left shift is performed during rounding. If after rounding $IE^{RIP} > Emax$, then overflow has occurred and an overflow value based on the sign and rounding mode is selected as shown in Table 1. Details on how this selection integrates with the rounding logic can be found in Subsection 3.5.

Detecting and avoiding underflow is handled similarly to the method described for overflow, but with the added complication of subnormal numbers. If $IE^{IP} < Emin$, then it is impossible to avoid underflow and a subnormal number or zero must be produced. Subnormal numbers are discussed in the next paragraph. If $IE^{IP} \geq Emin$ and $IE^{SIP} < Emin$, then underflow is avoided by decreasing the calculated *SLA* value from Equation 2 so that $IE^{SIP} = Emin$. In this case, it is always possible to perform this correction since $IE^{SIP} = IE^{IP} - SLA$. After rounding is performed, detecting underflow is handled by checking two cases resulting from the possible corrective left shift. First, if $IE^{RIP} < Emin$ and the fractional portion of the result is non-zero, or second, if $IE^{RIP} = Emin$ and the MSD of the result is zero and the fractional portion of the result is non-zero (indicating a corrective left shift should have occurred), then underflow has occurred and the corresponding flag is set.

As mentioned above, the need for a subnormal output is detected during exponent calculation if $IE^{IP} < Emin$. One design consideration is whether the subnormal result is calculated in hardware or software. In the proposed design, the detection of this case simply raises an output flag, causing a trap to software to complete the calculation of the subnormal result. This improves the design's area, latency, and frequency by removing the extra hardware needed to handle this case. If hardware subnormal result calculation is desired, however, it can be added to the algorithm by replacing the left shifter with a right-left barrel shifter and calculating a shift right amount (*SRA*) in parallel with the fixed-point multiplier using the following equation.

$$SRA = min((Emin - IE^{IP}), p + 2) \quad (5)$$

In this case, IE^{SIP} must also be modified to be $IE^{IP} + SRA$ when a subnormal result is detected. The same conditions can still be used to detect underflow.

3.4. Rounding

Rounding must be performed when all of the significant digits of the intermediate product do not fit within the *p* digits of the result's significand. The IEEE P754 draft standard specifies several rounding modes that must be supported. Each rounding mode and its associated condition(s) are listed in Table 1. In this table, the $\vee$ and $\wedge$ symbols represent the logical OR and AND operations respectively.

In the normal case without overflow or underflow, rounding is accomplished by selecting either the shifted intermediate product (SIP) truncated to *p* digits or its incremented value. These values are represented as TP^{+0} and TP^{+1}, respectively. Since the SIP is already in non-redundant form, a very simple direct decimal [17] incrementer is used to quickly increment the truncated value. Since a corrective left shift may need to be performed, the guard digit, g, also needs to be incremented. This may produce a carry into the upper portion of the shifted intermediate product, and this case is handled by the rounder's selection logic. Finally, to correctly handle all rounding modes, the design also preserves the digit to the right of the guard digit, henceforth called the round digit *r*.

One simplification within the rounding logic is that the proposed design does not need to contend with rounding overflow. Rounding overflow happens when the incrementing of the truncated intermediate product produces a carry-out of the MSD position. Due to the range of each operand and the method in which the shift amount is determined, rounding overflow cannot occur in this design. To learn more about why this is true, the reader is referred to [15].

To perform the rounding selection, several flags and indicators must be calculated. These values are all calculated in parallel with the incrementer to improve the latency of the overall design. First, the sticky bit, *sb*, is calculated from the lower 14 digits of the SIP using a standard tree of OR gates. Next, an indicator, *grsb*, is set whenever there are non-zero digits to the right of the decimal point. This is calculated as: $grsb = (g > 0) \vee (r > 0) \vee sb$. If the *grsb* indicator is set, then there is a corrective left shift when there is a leading zero in the truncated product TP^{+0}. The exception to this case is when we are rounding up and the shifted intermediate product is a zero followed by all nines. In this case, TP^{+1} is selected as the final result and the increment causes the MSD to become one, eliminating the need for a corrective left shift. This case is easily be detected by testing if the MSD of TP^{+0} is zero and the MSD of TP^{+1} is non-zero.

Next, the current rounding mode is used to set two round-up flags $ru^{cls==0}$ and $ru^{cls==1}$ which indicate if a round-up should be performed for the cases of no corrective left shift and a corrective left shift by one, respectively. The second column of Table 1 shows the computation of the $ru^{cls==0}$ value for each rounding mode based on the guard digit, the round digit, the sticky bit, the sign of the result, and the LSB of TP^{+0}. The $ru^{cls==1}$ value is calculated using similar equations in which the guard digit's LSB is

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

Rounding Mode	Condition for Selecting TP^{+1} (Non-overflow)	Product Override (Overflow) $s^P == 0$	$s^P == 1$
roundTiesToEven	$(g > 5) \vee ((g == 5) \wedge (l \vee (r > 0) \vee sb))$	$+\infty$	$-\infty$
roundTiesToAway	$g \geq 5$	$+\infty$	$-\infty$
roundTowardPositive	$!s^P \wedge ((g > 0) \vee (r > 0) \vee sb)$	$+\infty$	$-$Max Num.
roundTowardNegative	$s^P \wedge ((g > 0) \vee (r > 0) \vee sb)$	$+$Max Num.	$-\infty$
roundTowardZero	*none*	$+$Max Num.	$-$Max Num.

Legend: g=guard digit, r=round digit, s^P=sign of the product, sb=sticky bit, l=LSB of TP^{+0}

treated as the LSB of TP^{+0} and the round digit is treated as the guard digit.

Case 1: "No leading zeros, no corrective left shift"

MSD of TP^{+0} $!=$ 0 and MSD of TP^{+1} $!=$ 0

(a) $ru^{cls==0} == 0 \Rightarrow FP = TP^{+0}$

(b) $ru^{cls==0} == 1 \Rightarrow FP = TP^{+1}$

Case 2: "Leading zeros, corrective left shift"

MSD of TP^{+0} $==$ 0 and MSD of TP^{+1} $==$ 0

(a) $grsb == 0$

i. $IE^{SIP} == PE$ or $IE^{SIP} \leq Emin$ $\Rightarrow FP = TP^{+0}$

ii. $IE^{SIP} > PE$ and $IE^{SIP} > Emin$ $\Rightarrow FP = (TP^{+0} \ll 1) \,||\, g$

(b) $grsb == 1$ and $IE^{SIP} \leq Emin$

i. $ru^{cls==0} == 0 \Rightarrow FP = TP^{+0}$

ii. $ru^{cls==0} == 1 \Rightarrow FP = TP^{+1}$

(c) $grsb == 1, IE^{SIP} > Emin$, and $ru^{cls==1} == 0$ $\Rightarrow FP = (TP^{+0} \ll 1) \,||\, g$

(d) $grsb == 1, IE^{SIP} > Emin$, and $ru^{cls==1} == 1$

i. $g9 == 0 \Rightarrow FP = (TP^{+0} \ll 1) \,||\, (g+1)$

ii. $g9 == 1 \Rightarrow FP = (TP^{+1} \ll 1) \,||\, (g+1)$, note $g+1 == 0$

Case 3: "Zero followed by all nines"

MSD of TP^{+0} $==$ 0 and MSD of TP^{+1} $!=$ 0

(a) same as in case 2

(b) same as in case 2

(c) same as in case 2

(d) $grsb == 1, IE^{SIP} > Emin$, and $ru^{cls==1} == 1$ $\Rightarrow FP = TP^{+1}$

Figure 3. Rounding Scheme

Finally, the guard digit must be incremented and the possible carry-out of the guard digit indicated to the rounding selection logic. The incremented guard digit is used in the case of both a corrective left shift and round-up. A one-digit direct decimal adder is used to perform this increment and the carry-out is used as the flag *g9* to indicate that a carry-out as occurred.

Using the values for TP^{+0}, TP^{+1}, $ru^{cls==0}$, $ru^{cls==1}$, *grsb*, *g*, *g+1*, and *g9*, the algorithm presented in Figure 3 is used to realize IEEE P754 compliant rounding, based on the algorithm originally presented in [15]. The algorithm contains three main cases based on the MSDs of TP^{+1} and TP^{+0}. The fourth case, in which the MSD of TP^{+0} is non-zero and the MSD of TP^{+1} is zero, cannot occur. The proposed parallel multiplier design calculates all of the above flags and indicators in parallel with the increment producing TP^{+1}. In addition, some of the comparisons and selection conditions found in Figure 3 are calculated in parallel with the increment. This significantly reduces the rounding latency and the latency of the overall design.

3.5. Special Case Handling

There are four exceptions that may be signaled during multiplication; invalid operation, overflow, underflow, and inexact. Overflow and underflow are detected by the design during rounding as described in Subsection 3.3. In either the overflow or underflow case, the inexact exception is also raised to indicate essential digits have been lost. When overflow is detected, the IEEE P754 standard specifies the selection of either the largest normal number or canonical infinity as the output based on the current rounding mode and product sign, as given in Table 1. This selection is performed after rounding, as discussed later in this section.

The invalid operation exception is signaled only when zero and infinity are multiplied or when either operand is a signaling NaN. In either case, the invalid exception flag is raised and a quiet NaN is produced as the result. Including invalid operations, there are five cases that create a quiet NaN at the output. In the case of zero and infinity being multiplied, a quiet NaN with a significand of zero is produced. Otherwise the quiet NaN's significand is generated from the input signaling NaN or the input quiet NaN is passed to the output with preference given to signaling NaNs. A signaling NaN is never produced by the multiplier. To allow the passing of NaNs from the input to the output, the proposed design uses multiplexing within the decode

stage to override one of the input operands of the fixed point multiplier to a value of one. This passes the other operand through the fixed point multiplier unmodified without the need for a bypass path. The rounding logic is modified to ensure the passed value does not get incremented or shifted.

The above exceptional cases are handled by two multiplexors following the rounder and occur before the final IEEE P754 encoding step. The first multiplexor selects between infinity, zero, and the largest normal number based on the various exception flags and the sign of the result. This is done in parallel with the rounding multiplexing. A final multiplexor then selects between the rounded significand and the output from the exception multiplexor. This final value is input into the encoding logic. Since nearly all exception flags except overflow and underflow are calculated in parallel with the rounding logic, this exception logic has only a minor impact on the design's latency.

4. Results

4.1. Verification and Synthesis

To test and analyze the proposed decimal floating-point multiplier, register transfer level models of the design were coded in Verilog, including RTL models of the fixed-point multiplier from [1]. To validate the designs, large numbers of random vectors were applied to simulate the design with ModelSim. For the fixed-point multiplier, over 500,000 random test cases were used to verify the design along with hand-picked vectors to test all corner cases. For the floating-point design, over 500,000 test cases were used to test all available rounding modes and exceptions. A large number of these test cases were directed-random cases created by IBM's FPgen tool [10]. Directed tests soon to be available at [9] were also simulated. A description of how these test cases were generated can be found in [16].

To analyze the delay and area of our design, the Verilog models were synthesized using LSI Logic's gflxp 0.11 μm CMOS standard cell library and Synopsys Design Compiler Y-2006.06-SP1. In addition, the Synopsys's DesignWare IP library was used extensively to further improve the results. Although using the DesignWare library reduces the design's portability, it more accurately models the optimizations that would be made if this design was used in industry. The design was synthesized for a large range of pipeline depths to explore tradeoffs in terms of latency, area, and delay. This synthesis was performed using Synopsys's auto-pipelining feature, which introduces a small amount of variability in the results. The synthesis results are given in Table 2. The fan-out-of-four (FO4) inverter delay values are calculated by dividing the critical path delay by 55 ps, which is the delay of an inverter driving four similar inverters in the cell library. An area and delay breakdown giving the approximate contribution of each major component in a combinational version of the design is given in Table 3.

Table 2. Area and Delay for Multiplier Designs

Mult Design	Delay		Area	
	ps	FO4	Cells	μm^2
Parallel Mult				
Combinational	4440	80.7	322,493	651,435
1 Stages	4420	80.4	334,400	675,488
2 Stages	2360	42.9	349,429	705,846
3 Stages	1810	32.9	364,594	736,479
4 Stages	1540	28.0	372,082	751,605
5 Stages	1310	23.8	364,829	736,954
6 Stages	1140	20.7	367,588	742,527
7 Stages	1090	19.8	376,199	759,922
8 Stages	980	17.8	399,636	807,264
9 Stages	930	16.9	395,953	799,825
10 Stages	880	16.0	409,156	826,495
11 Stages	850	15.5	412,795	833,845
12 Stages	820	14.9	427,094	862,729
Sequential Mult[a]	850	15.45	117,627	237,607

[a] Multiplier design from [15] with 21 cycle latency and a throughput of 1/21 multiplications per cycle

Table 3. Combinational Design Area and Delay Breakdown

Component	Delay Breakdown		Area Breakdown	
	ps	%	μm^2	%
Fixed-point Multiplier	2990	67.3%	326,841	50.2%
Round logic	810	18.2%	14,125	2.2%
Left shift	280	6.3%	17,414	2.7%
Datapath [a]	360	8.1%	26,605	4.1%
Interconnect	N/A	N/A	270,243	41.5%
Entire Design	4440	100%	651,435	100%

[a] Includes LZD, SLA generation, exponent logic, etc.

4.2. Analysis and Optimizations

Additional data is provided in Table 2 for a sequential floating-point design that complies with IEEE P754 and is implemented using the same tools and technology [15]. This sequential design produces one multiplication result every 21 cycles in the normal case. An 11-stage version of our design has similar critical path delay, significantly reduced latency, and one result per cycle throughput while

incurring a substantial 371% increase in area. Our design also allows tradeoffs between latency, area, and clock speed by selecting a different number of pipeline stages. The sequential design, due to its iterative algorithm, does not have this flexibility.

Several optimizations were investigated during the design of the multiplier. Other rounding algorithms were explored, such as the use of injection-based rounding [7]. Even though these schemes may offer small delay improvements over the rounding scheme in Subsection 3.4, their impact on the final latency and delay of our design is small due to the dominant delay of the fixed-point multiplier. A method to pre-calculate the sticky-bit in parallel with the fixed-point multiplier was also considered, however this method did not produce improved results as compared to a simple OR gate implementation after the fixed-point multiplier. We also explored performing the shift operation before the carry-propagate adder in the fixed-point multiplier, but any delay or area savings from using a smaller 16-digit CPA is lost in the additional complications of calculating the sticky bit from a redundant carry-save representation and the two 128-bit shifters. We leave these and other optimizations to be examined further in future work.

5. Summary

In this paper, a novel IEEE P754 compliant parallel decimal floating-point multiplier design was presented based on a parallel fixed-point multiplier and a previous sequential floating-point multiplier. Several components of the design were described, including the fixed-point multiplier, exponent calculation, rounding, and exception handling. Novel elements include early shift amount calculation, highly parallel datapath, and exception value pass-through. The design was implemented using Verilog and verified using extensive directed-random vectors. Synthesized standard cell estimates are presented for several design points. These show an 11-stage pipelined version of the presented design significantly outperforms a sequential IEEE P754 floating-point multiplier in terms of both latency and throughput, while maintaining a similar clock frequency.

Acknowledgment

This work is sponsored in part by International Business Machines. The authors are grateful to Alvaro Vazquez, Elisardo Antelo, and Paolo Montuschi for their parallel fixed-point multiplier design used in our DFP multiplier.

References

[1] A. Vazquez, E. Antelo, and P. Montuschi. A New Family of High–Performance Parallel Decimal Multipliers. In *18th IEEE Symposium on Computer Arithmetic*, pages 195–204, June 2007.

[2] 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, 1987.

[3] 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.

[4] M. F. Cowlishaw. Densely Packed Decimal Encoding. *IEEE Proceedings – Computers and Digital Techniques*, 149(3):102–104, May 2002.

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

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

[7] G. Even and P.-M. Seidel. A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication. *IEEE Transactions on Computers*, 49(7):638–650, 2000.

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

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

[10] 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.

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

[12] R. D. Kenney, M. J. Schulte, and M. A. Erle. A High-Frequency Decimal Multiplier. *International Conference on Computer Design*.

[13] T. Lang and A. Nannarelli. A Radix-10 Combinational Multiplier. In *Proc. of 40th Asilomar Conference on Signals, Systems, and Computers*, pages 313–317, Oct 2006.

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

[15] M. A. Erle, M. J. Schulte, and B. J. Hickmann. Decimal Floating-Point Multiplication Via Carry-Save Addition. In *18th IEEE Symposium on Computer Arithmetic*, pages 46–55, June 2007.

[16] R. M. M. Aharoni and A. Ziv. Solving Constraints on the Intermediate Result of Decimal Floating-Point Operations. In *18th IEEE Symposium on Computer Arithmetic*, pages 38–45, June 2007.

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

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