Decimal Floating-Point Adder and Multifunction Unit with Injection-Based Rounding

Liang-Kai Wang and Michael J. Schulte
Department of Electrical and Computer Engineering
University of Wisconsin-Madison, Madison, WI 53705, USA
lwang@cae.wisc.edu, schulte@engr.wisc.edu

Abstract

Shrinking feature sizes gives more headroom for designers to extend the functionality of microprocessors. The IEEE 754R working group has revised the IEEE 754-1985 Standard for Binary Floating-Point Arithmetic to include specifications for decimal floating-point arithmetic and IBM recently announced incorporating a decimal floating-point unit into their POWER6 processor. As processor support for decimal floating-point arithmetic emerges, it is important to investigate efficient algorithms and hardware designs for common decimal floating-point arithmetic algorithms. This paper presents novel designs for a decimal floating-point adder and a decimal floating-point multifunction unit. To reduce their delay, both the adder and the multifunction unit use decimal injection-based rounding, a new form of decimal operand alignment, and a fast flag-based method for rounding and overflow detection. Synthesis results indicate that the proposed adder is roughly 21% *faster and* 1.6% *smaller than a previous decimal floating-point adder design, when implemented in the same technology. Compared to the decimal floating-point adder, the decimal floating-point multifunction unit provides six additional operations, yet only has* 2.8% *more delay and* 9.7% *more area.*

1 Introduction

Binary floating-point arithmetic is usually sufficient for scientific and statistics applications. However, it is not sufficient for many commercial applications and database systems, in which operations often need to mirror manual calculations. Therefore, these applications often use software to perform decimal floating-point arithmetic operations. Although this approach eliminates errors due to converting between binary and decimal numbers and provides decimal rounding to mirror manual calculations, it results in long latencies for numerically intensive commercial applications. Because of the growing importance of decimal floating-point arithmetic, specifications for it have been added to the draft revision of the IEEE-754 Standard for Floating-Point Arithmetic (IEEE P754) [1].

Previous research in the area of decimal addition has focused on decimal fixed-point addition [4, 6, 10–13, 16, 18]. For example, in [13], Schmookler et al. develop a method for high-speed decimal addition that incorporates the weight of each bit in a decimal digit and the carry into the digit to compute a final sum digit. Busaba, Bultmann, and Haller propose combined decimal and binary adders using pre-sum and pre-selection logic [4, 6, 10]. Svoboda in [18], Sacks-Davis in [12], Shirazi et al. in [16], and Nikmehr et al. in [11] implement decimal addition based on signed-digit arithmetic with digit sets, ranging from [-6, 6] to [-9, 9].

Only a few previous papers present techniques for decimal floating-point addition [3, 7, 19]. The adder designs by Cohen et al. [7] and Bolenger et al. [3] have long latencies and produce one result digit each cycle. In [19], the authors present the first IEEE P754-compliant decimal floating-point adder. Their design introduces novel techniques for significand alignment, result correction, and rounding. IBM recently announced incorporating a decimal floating-point unit into their POWER6 processor [15].

In this paper, we present novel designs for a decimal floating-point adder and a decimal floating-point multifunction unit. Compared to the adder presented in [19], our adder has 21% less delay and 1.6% less area when implemented in the same technology. Our decimal floating-point multifunction unit is based on our decimal adder and performs eight decimal operations defined in IEEE P754; addition, subtraction, compare, minNum, maxNum, quantize, sameQuantum, and roundToIntegral. Synthesis results show that the proposed multifunction unit has only 2.8% more delay and 9.7% more area than our decimal floating-point adder with injection-based rounding. The decimal floating-point adder and multifunction unit presented in this

Format name	decimal32	decimal64	decimal128
Total storage width	32	64	128
Combination field width $(w + 5)$	11	13	17
Trailing significand field (t)	20	50	110
Total significand digits (p)	7	16	34
Exponent bias	101	398	6176

Table 1. Parameters for Different Decimal Interchange Formats.

paper support 64-bit decimal floating-point operands, but the techniques presented in this paper could easily be extended to handle other operand sizes.

The rest of the paper is organized as follows. Section 2 gives an overview of decimal arithmetic in IEEE P754. Section 3 describes our proposed decimal floating-point adder with injection-based rounding. Section 4 discusses the decimal floating-point multifunction unit. Section 5 presents synthesis results for our proposed adder and multifunction unit and for the adder from [19]. Section 6 gives our conclusions.

In the rest of this paper, CX_Y and EX_Y are the significand and the exponent of a decimal number, respectively. X is either A or B to denote the operand, and the subscript "$_Y$" is a digit that denotes the output of different modules. "$(N)_Z^T$" refers to the T^{th} bit in digit position, Z, in a number, N, where the least significant bit and the least significant digit have index 0. For example, $(CA_1)_2^3$ is the third bit of the second BCD digit in CA_1.

2 Overview of Decimal Floating-Point Arithmetic in the IEEE P754 Standard

Figure 1 shows the basic decimal interchange format specified in IEEE P754. S is the sign bit and G is a combination field that contains the exponent, the most significant digit of the significand, and the encoding classification. The rest of the significand is stored in the Trailing Significand Field, T, using either the Densely Packed Decimal (DPD) encoding or the Binary Integer Decimal (BID) encoding. The DPD encoding represents every three consecutive decimal digits in the decimal significand using 10 bits, and the BID encoding represents the entire decimal significand in binary. Table 1 shows the parameters for each decimal interchange format, where the total number of significand digits in each format corresponds to the format's precision, p. Our proposed decimal floating-point adder and multifunction unit operate on numbers in the decimal64 format with DPD encoding.

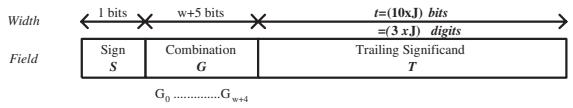

Figure 1. Decimal Floating-Point Interchange Format

The significand of a decimal number is not normalized, which means that a single decimal floating-point number may have multiple representations. A set of these equivalent decimal numbers is called a **cohort**. Because of this characteristic, the standard defines the Preferred Representation Exponent, which refers to a required exponent (and implicitly the significand) after a decimal operation. For example, in decimal addition, the standard selects the cohort member whose exponent equals the smaller exponent of the two input operands if the result is exact. If the result is not exact, the result is rounded to p digits. In addition to the specification of the decimal interchange formats, IEEE P754 also specifies operations on decimal numbers, including two decimal-specific operations; sameQuantum and quantize, which are discussed in Section 4.1.

3 Decimal Floating-Point Adder

This section presents our decimal floating-point adder, which uses a parallel method for decimal significand alignment and a Kogge-Stone parallel prefix network for significand addition and subtraction. It also applies decimal variations of the injection-based rounding method [9] and the flagged prefix network [5] to decrease the latency of rounding and overflow detection. The decimal floating-point adder supports all the rounding modes and appropriate exceptions specified in IEEE P754 and all the rounding modes specified in the Java BigDecimal library [17].

Figure 2 shows a high-level block diagram of our proposed decimal floating-point adder. The 'Forward Format Conversion Unit' takes the two IEEE-encoded operands, A and B, and the operation, and produces the sign bits, SA_1 and SB_1, BCD significands, CA_1 and CB_1, biased exponents, EA_1 and EB_1, and effective operation, EOP (not shown in the figure). The 'Operand Alignment Calculation and Swapping Unit' (OACSU) takes these values and computes the result's temporary exponent, ER_1, the right shift amount, RSA, and the left shift amount, LSA. It also swaps the significands if $EB_1 > EA_1$. The two significands after swapping are denoted as CA_S and CB_S. Next, two 'Decimal Barrel Shifters' take these results and perform operand alignment on CA_S and CB_S. The two shifted significands, CA_2 and CB_2, are then corrected in the 'Precorrection Unit'. Based on the EOP signal and the pre-

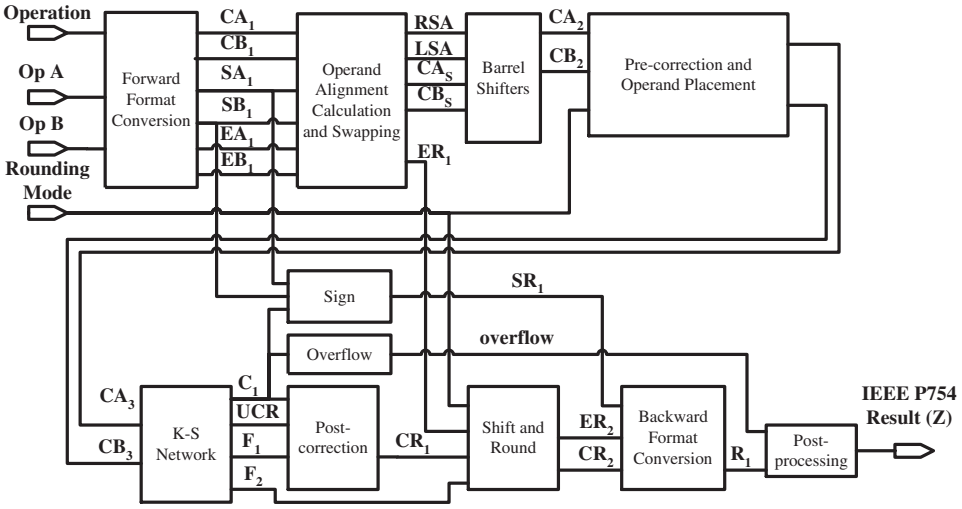

Figure 2. Block Diagram of the Proposed Decimal Floating-Point Adder

vailing rounding mode, the 'Pre-correction Unit' prepares the BCD operands for addition or subtraction and inserts a value needed for injection-based rounding. The corrected significands, CA_3 and CB_3, are then fed into the Kogge-Stone (K-S) network, which produces an uncorrected result, UCR, a digit-carry vector, C_1, and flag vectors, F_1 and F_2. After this, the 'Post-correction Unit' converts UCR back into the BCD encoding to produce CR_1. If needed, the 'Shift and Round Unit' shifts and rounds CR_1 to produce the result's significand, CR_2, and adjusts the temporary exponent, ER_1, to produce the result's exponent, ER_2. Simultaneously, the 'Sign Unit' and the 'Overflow Unit' compute the result's sign bit, SR_1, and the overflow signal. The result's values, CR_2, ER_2, and SR_1, are combined to generate the IEEE-encoded result in the 'Backward Format Conversion Unit'. This result and the original input operands are examined in the 'Post-processing Unit' to determine if a special result is needed, which happens if either one or both of the operands are Not-a-Number (NaN) or infinity. Further details on each of these units are provided below.

3.1 Forward Format Conversion and Operand Alignment Calculation and Swapping

The core of our decimal floating-point adder operates on BCD significands. Therefore, converters are first employed to extract the DPD-encoded significands, binary exponents, and sign bits from both IEEE-encoded operands. The two DPD-encoded significands are simultaneously converted to BCD format using the equations given in [1, 8]. Once unpacked, the two resulting significands are swapped if $EB_1 > EA_1$ and the temporary result exponent, ER_1, is determined. The two significands after swapping are denoted as CA_S and CB_S where the subscript "$_S$" refers to *Swapped*. The number of leading zeros in the significand with the larger exponent, CA_S, is denoted as LA_S. In parallel with swapping the operands, the effective operation (EOP) is determined by the Boolean equation $EOP = SA_1 \oplus SB_1 \oplus Operation$, where EOP and $Operation$ are zero for addition and one for subtraction.

Decimal operand alignment is more complex than its binary counterpart because decimal numbers are not normalized. This leads to both left and right shifts to ensure the rounding location is in a fixed digit position. To correctly adjust both operands to have the same exponent, the following computations are performed:

$$LSA = \min(|EA_1 - EB_1|, LA_S)$$
$$RSA = \min(\max(|EA_1 - EB_1| - LA_S, 0), 19)$$
$$ER_1 = EA_S - LSA$$

The above equations produce a left shift amount, LSA, which indicates by how many digits CA_S should be left shifted. LSA is equal to the absolute value of the exponent difference, $|EA_1 - EB_1|$, but is limited to LA_S digits so that the left-shifted significand, CA_2, does not have more than 16 digits. The RSA value indicates by how many digits CB_S should be right shifted in order to guarantee that both numbers have the same exponent, ER_1, after significand alignment. RSA is limited to 19 digits, since the right-shifted significand, CB_2, contains 16 digits plus guard and round digits and a sticky bit, as explained in Section 3.2. The temporary result exponent, ER_1 is simply the larger exponent, EA_S, after it has been adjusted to compensate for the left shift amount, LSA.

The technique used in [19] to perform operand swap-

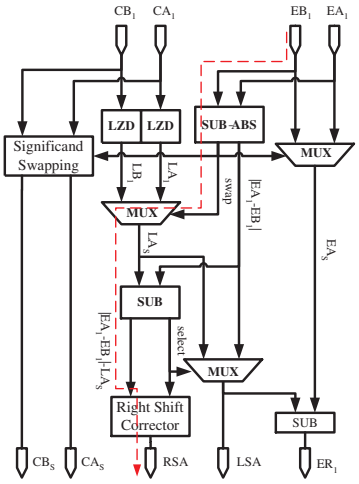

Figure 3. Operand Alignment Calculation and Swapping Unit

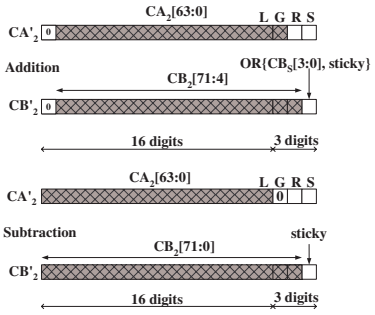

Figure 4. Operand Placement for Decimal Addition and Subtraction.

ping and alignment computation is to subtract EB_1 from EA_1 and use the sign of the result to determine which operand has the larger exponent. With this technique, if $sign(EA_1 - EB_1)$ is one, then B has the larger exponent and the operands should be swapped; otherwise the operands should not be swapped. After operand swapping, the significand of the number with the larger exponent is examined to determine its leading zero count. With this approach, leading zero detection occurs after operand swapping, which leads to only one leading zero detector but a long delay.

To reduce the delay, our design uses a binary absolute value unit [5] to compute $|EA_1 - EB_1|$ and $swap = sign(EA_1 - EB_1)$ and in parallel performs leading zero detection on both CA_1 and CB_1 to produce LA_1 and LB_1. If $swap$ is one, then $CA_S = CB_1$, $LA_S = LB_1$, and $EA_S = EB_1$; Otherwise, $CA_S = CA_1$, $LA_S = LA_1$, and $EA_S = EA_1$. LA_S is then subtracted from $|EA_1 - EB_1|$ to compute RSA and $select = sign(|EA_1 - EB_1| - LA_S)$, which is used to select the value for LSA and ensures RSA is greater than zero. RSA is limited to a value between 0 and 19 by the Right Shift Corrector and in parallel $ER_1 = EA_S - LSA$ is computed. This approach is shown in Figure 3, where the dashed line indicates the critical delay path of the unit. Our synthesis results indicate that this approach reduces the critical path delay of the 'Operand Alignment Calculation and Swapping Unit' by roughly 41% and increases its area by roughly 4.8% compared to the design in [19].

Table 2. Injection Values Based on Different Rounding Modes ('X' is don't-care).

$Sign_{inj}$	Rounding Modes	Injection Value (R, S)
X	RoundTowardZero	(0 , 0)
X	RoundTiesToAway	(5 , 0)
X	RoundTiesToZero	(4 , 9)
X	RoundTiesToEven	(5 , 0)
-	RoundTowardPositive	(0 , 0)
+	RoundTowardPositive	(9 , 9)
-	RoundTowardNegative	(9 , 9)
+	RoundTowardNegative	(0 , 0)
X	RoundAwayZero	(9 , 9)

3.2 Operand Alignment and Precorrection

After computing the left and right shift amounts, two decimal barrel shifters, which shift by multiples of four bits, perform the operand alignment. The significands after alignment are denoted as $CA_2 = left\_shift(CA_S, LSA)$ and $CB_2 = right\_shift(CB_S, RSA)$. As noted previously, CA_2 is 16 digits, and CB_2 is 16 digits plus a guard digit, G, a round digit, R and a sticky bit, S, as shown in Figure 4. In this figure, L denotes the least significant digit (LSD). The sticky bit is generated from the shift amount and CB_2, where the bits of the shift amount are used to create masks. The mask bits are ANDed with several bits of CB_2, so that only bits in CB_2 that would be shifted to the right of the round digit contribute to the sticky bit. The output bits from the AND operation are then ORed together to produce the sticky bit. With decimal floating-point arithmetic in IEEE P754, it is possible to have a zero operand with an exponent that is greater than the exponent of another nonzero operand. In this case, neither operand is shifted.

Once shifted, an injection value based on the sign bit and prevailing rounding mode is inserted into the Round

and Sticky digit positions of CA_2 to form CA'_2, which is a 19-digit BCD number. The injection value, shown in Table 2, is determined by equations similar to those developed for binary floating-point addition [14] and is used to facilitate correct rounding. CB_2 may be inverted depending on the effective operation (EOP). In Table 2, $Sign_{inj}$ is the temporary sign of the result, which assumes the result after the K-S network is positive when rounding is performed. This assumption is valid because if the result from the K-S network is negative, there is no right shift on CB_S, so no rounding is needed. The sign bit used to select the injection value is computed as:

$$Sign_{inj} = \begin{array}{l} (\overline{EOP \wedge swap}) \wedge SA_1 \vee (EOP \wedge swap) \wedge \\ (Operation \oplus SB_1) \end{array} \quad (1)$$

Based on EOP, the modified CA_2 and CB_2 are placed in different digit positions before entering the K-S network, as shown in Figure 4 .

As shown in Figure 4, both operands are placed starting from the second to the most significant digit (MSD) of the 19-digit wide datapath for addition and from the most significant digit for subtraction. This placement allows the 16-digit final result to be selected from the 17 more significant digits and simplifies the alignment of injection and injection correction values such that these values are placed in the same locations for both effective addition and subtraction. The injection value is inserted into the two least significant digits, R and S, based on the prevailing rounding mode and $Sign_{inj}$. The injection value is inserted on all addition/subtraction-related operations except when the EOP is subtraction and no right shift is performed on CB_2. In this case, since rounding cannot occur and the result from the K-S network may be negative, inserting the injection value might unnecessarily complicate the post-correction logic. To avoid this condition, another signal, $flushing$, is generated to clear the injection value. This signal is computed as $flushing = EOP \wedge (RSA \neq 0)$ The operands after placement are denoted as CA'_2 and CB'_2 and both are 19 digits.

After the injection value is inserted into the datapath, both operands are pre-corrected in order to generate the proper carry-out of each digit. The equations implemented by the 'Pre-correction Unit' are:

$$(CA_3)_i = \begin{cases} (CA'_2)_i + 6 & \text{EOP} \equiv add \\ (CA'_2)_i & \text{otherwise} \end{cases}$$

$$(CB_3)_i = \begin{cases} (CB'_2)_i & \text{EOP} \equiv add \\ \overline{(CB'_2)_i} & \text{otherwise} \end{cases} \quad (2)$$

$$i = 0...18$$

where $\overline{(CB'_2)_i}$ is the fifteen's complement of $(CB'_2)_i$ and is obtained by inverting each bit of $(CB'_2)_i$.

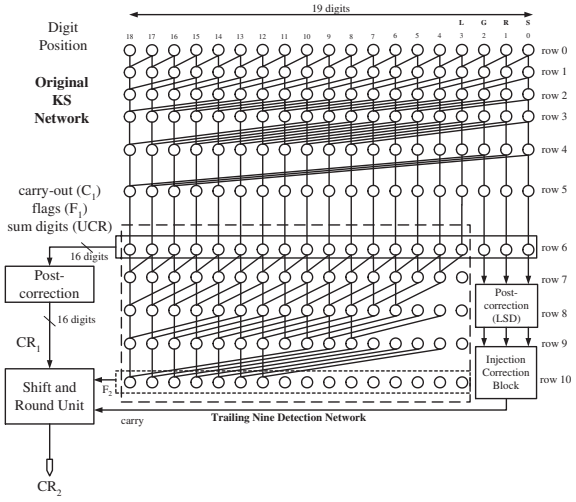

Figure 5. Conceptual View of the Flag-based Logic and the Kogge-Stone Network.

With effective addition, each digit $(CA'_2)_i$ is incremented by 6 such that in each digit position the operation performed is $(CA'_2)_i + 6 + (CB'_2)_i + (C_1)_i$, where the digit sum is within the range of [6, 25] and $(C_1)_i$ is the carry into digit i. With effective subtraction, the operation performed at each digit is $(CA'_2)_i + 15 - (CB'_2)_i + (C_1)_i = (CA'_2)_i + 6 + (9 - (CB'_2)_i) + (C_1)_i$, and the digit sum is also in range [6, 25]. Having each digit sum $\in [6, 25]$ helps generate correct carries using the Kogge-Stone network and simplifies converting the result back to BCD. More details on how the result from the K-S network is converted back to BCD are given in Section 3.4.

3.3 Decimal Kogge-Stone Network

Because both operands are corrected, a binary Kogge-Stone (K-S) network can be used to generate the proper carry into each digit. In addition to the flags used in the post-correction step (i.e. F_1 in this paper) [19], two more sets of flags, $flagADD$ and $flagSUB$, which are used in the 'Shift and Round Unit' are generated. These flags are used for addition and subtraction, respectively, to avoid another carry-propagate addition when the most significant digit (MSD) of CR_1 is nonzero. For example for $p = 7$, if $CA_3 = 0\_9999999\_99$, and $CB_3 = 0\_0039999\_91$ and we perform addition with round toward positive infinity, CR_1 becomes $1\_0039999\_90$ and has an MSD of 1. Examining the result indicates that there are three consecutive nines starting from the least significant digit (LSD) (the rightmost two nine's are discarded when $p = 7$). Therefore, we increment the four least significant digits and the final result becomes 1004000×10^1 after shifting and rounding. De-

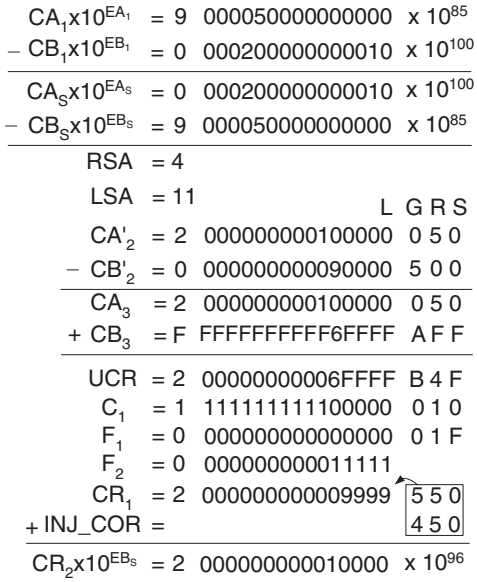

Figure 6. Example of Decimal Subtraction with Injection-Based Rounding and Trailing-nine Detection.

termining which digits need to be incremented is performed by a method known as trailing nine detection. It is important to note that the trailing-nine detection is only needed if $EOP \equiv ADD$ or $EOP \equiv SUB$ and $CA_3 - CB_3$ is positive. If $CA_3 - CB_3$ is negative, there is no need to perform rounding and trailing-nine detection since the final result guarantees to be less than 17 digits.

Figure 5 illustrates how the original K-S network is extended to detect trailing nines. The traditional injection-based rounding method uses conditional adders to compute the uncorrected sum and the uncorrected sum plus one and then uses the MSDs of these values and the carry into the LSD of the uncorrected sum to select the proper sum. To reduce area, our adder instead uses the flagged-prefix method [5] to compute the uncorrected sum and the uncorrected sum plus one. Since the value generated in the K-S network is not in BCD format, the bits of F_2 are generated by observing both the sum digits, $(UCR)_i$, and the carry-out bits, $(C_1)_{i+1}$ of the 16 most significant digits. An example of decimal floating-point subtraction is shown in Figure 6, where F_1 is the flag used in the 'Post-correction Unit', described in Section 3.4. To generate the F_2 flags for trailing nine detection, UCR can be examined for trailing F's or CR_1 can be examined for trailing nines starting from the LSD. Examining CR_1 only requires one set of flags, but computing these flags is on the critical path. Therefore, our design computes the F_2 flags based on UCR. Although this approach decreases the delay, two sets of flags, $flagADD$ and $flagSUB$, are needed for addition and subtraction, respectively.

Although there are several extra stages in the K-S network for trailing-nine detection, these stages work in parallel with the 'Post-correction Unit', and 'Injection Correction Unit' and therefore the trailing-nine detection is not on the critical path. The equations used in row 6, and rows 7-10 of the K-S network for trailing-nine detection are:
Row 6:

$$ADD : (flagADD_0)_i = \begin{array}{l}((UCR)_i \equiv 15) \vee \\ ((UCR)_i \equiv 9) \wedge \\ \big((C_1)_{i+1} \equiv 1\big)\end{array}$$
$$SUB : (flagSUB_0)_i = ((UCR)_i \equiv 15)$$

where $(C_1)_{i+1}$ is the carry-out bit of digit position i.
Rows 7-10 ($1 \leq x \leq 4$) :

$$(flagADD_x)_i = \begin{array}{l}(flagADD_{x-1})_i \wedge \\ (flagADD_{x-1})_{i-2^{x-1}}\end{array}$$
$$(flagSUB_x)_i = \begin{array}{l}(flagSUB_{x-1})_i \wedge \\ (flagSUB_{x-1})_{i-2^{x-1}}\end{array}$$
$$F_2 = \begin{cases} flagADD_4 & EOP \equiv ADD \\ flagSUB_4 & EOP \equiv SUB \end{cases}$$

Preliminary synthesis results of this method, compared to the Kogge-Stone network in [19], which only has one set of flags for the 'Post-correction Unit', shows only a 13.7% increase in area. Thus, the proposed injection-based method with trailing-nine detection can potentially use less area than the traditional injection-based method with a minor increase in delay. An even more aggresive design can use the End-Around-Carry (EAC) technique or its variation, presented in [2].

3.4 Post-Correction and Shift and Round

The temporary result generated from the Kogge-Stone network requires a post-correction unit to convert the uncorrected result, UCR back to BCD to produce CR_1. The rules for performing this correction are defined below:

Rule 1: Rule enforced when performing an effective addition operation:
Add 1010 (correction of -6) to $(UCR)_i$ in digit **i** *for which $(C_1)_{i+1}$ is 0*

Rule 2: Rule enforced when performing an effective subtraction operation:
If (MSB of $C_1 \equiv 1$) //the sign of the result is positive

1. Invert bits in UCR for which the corresponding bit in F_1 is one. This increments UCR.
2. Add '1010' to the above result in digit **i** for which $(C_1)_{i+1} \oplus (F_1)_i^3 \equiv 0$

Else // the sign of the result is negative

1. Invert all sum bits

2. Add '1010' to the above result in digit **i** for which $(C_1)_{i+1} \equiv 1$

Rule 1 is straightforward, since the pre-correction value is simply subtracted from each sum digit where no carry-out is generated from that digit position. For Rule 2, if the result is positive, UCR needs to be incremented by one since a nine's complement is performed on CB_2' in the 'Pre-correction Unit'. UCR is quickly incremented by inverting the bits in UCR for which the corresponding bit in F_1 is one. Because F_1 is generated in the K-S network, this action is easily performed using a row of exclusive-or gates. Next, if the most significant flag bit, $(F_1)_i^3$ and the carry-out, $(C_1)_{i+1}$, of digit position i are the same, then $(CA_3)_{i:0} < (CB_3)_{i:0}$. Therefore, a value of six should be subtracted from the sum digit, which is equivalent to adding a value of ten to the digit position. Similarly, if the result is negative, we first invert all sum bits such that $CR_1 = CB_3 - CA_3$. Next, if $(C_1)_{i+1}$ is one, $(CB_3)_{i:0} < (CA_3)_{i:0}$, and a value of six is subtracted from the sum digit at digit position i.

If needed, the 'Shift and Round Unit' rounds the corrected result, CR_1, from the 'Post-correction Unit' and shifts it right by one digit. Although a value has been injected to simplify this unit, similar to the injection-based method in binary arithmetic, it is still necessary to have an injection correction step if the most significant digit (MSD) of CR_1 is non-zero. In this case, a second correction value is added to CR_1 as shown in Table 3. Adding the injection correction value from Table 3 to the injection value from Table 2 gives the overall injection value needed when the MSD of CR_1 is non-zero.

As illustrated in Table 3, there are only two non-zero injection correction values, and S is always zero for injection correction. Since the injection correction value is only needed if the most significant digit (MSD) of CR_1 is non-zero, it is impossible to have another carry from the most significant digit due to adding the injection correction value. To avoid the carry propagation network needed to add the injection correction value, F_2 flags, which are generated in the K-S network, are used to increment CR_1, if needed.

3.5 Overflow, Sign, Backward Format Conversion, and Post-processing

Overflow occurs when the addition or subtraction of two operands exceeds the maximum representable value in the destination format. Typically, the adder needs to check the carry from the MSD after incrementing the corrected result to see if an overflow occurs. With the injection-based rounding method, however, since the injection correction value does not generate another carry from the MSD, the overflow signal can be generated by examining the final exponent from the operand alignment unit, ER_1, and the MSD of the corrected result, CR_1. The 'Overflow Unit', also generates a signal to determine if the final result should be infinity or the maximum representable value of the destination format, based on the rounding mode and the sign of the result. Using this signal and the overflow flag, the final result can be modified, if needed, in the 'Post Processing Unit'.

Table 3. Injection Correction Values Based on Different Rounding Modes.

$Sign_{inj}$	Rounding Modes	Injection Correction Value (G, R, S)
X	RoundTowardZero	(0, 0, 0)
X	RoundTiesToAway	(4, 5, 0)
X	RoundTiesToZero	(4, 5, 0)
X	RoundTiesToEven	(4, 5, 0)
-	RoundTowardPositive	(0, 0, 0)
+	RoundTowardPositive	(9, 0, 0)
-	RoundTowardNegative	(9, 0, 0)
+	RoundTowardNegative	(0, 0, 0)
X	RoundAwayZero	(9, 0, 0)

The sign bit of the result is determined by several factors. Due to space limitations, we only show the normal case in Equation (3) for when no special cases or exceptions occur.

$$Sign_{Normal} = \begin{array}{l}(\overline{EOP} \wedge SignA) \vee (EOP \wedge \\ (\overline{swap} \oplus SignA \oplus Carry_{MSD}))\end{array} \quad (3)$$

Since the sign bit is necessary in several other modules such as the 'Overflow Unit' and the 'Shift and Round Unit', we need to determine its value as soon as possible. To quickly determine the sign of the result, all the equations for the special cases are duplicated and the carry out of the MSD from the K-S network is used to select the correct sign bit.

The 'Backward Format Conversion Unit' encodes the sign bit, the exponent bits, and the significand digits to form the IEEE-encoded result. Finally, the 'Post-processing Unit' handles special input operands in IEEE P754, such as infinity, signaling, and quiet NaNs, and results that trigger exceptions, such as overflow. To comply with the standard, both inputs are propagated to this unit (not shown in Figure 2). In general, underflow occurs when the final result is subnormal and inexact [1]. Although it is possible to have a subnormal result when performing decimal floating-point addition and subtraction, such a subnormal result cannot also be inexact. Therefore, underflow does not occur when performing decimal floating-point addition or subtraction. It also does not occur for any of the other operations performed by the decimal multifunction unit described in Section 4.

3.6 Design Comparisons

There are some major differences between the proposed decimal floating-point adder and the design presented in [19]. First, the proposed design in parallel computes $|EA_1 - EB_1|$ and performs leading zero detection on CA_1 and CB_1 to reduce the overall delay. Second, it uses a decimal injection-based rounding method to reduce the length of the critical path in the 'Shift and Round Unit'. Third, in addition to the flags for the post-correction used in [19], there are two extra sets of flags, $flagADD$ and $flagSUB$, to more quickly increment the corrected result and generate the overflow flag via trailing-nine detection. There are also a few other minor optimization including the internal use of BCD encoding instead of excess-3 encoding, which leads to a simpler circuit in the 'Pre-correction Unit' and the more efficient placement of the corrected operands for addition and subtraction to simplify the design of the 'Shift and Round Unit'. A quantitative comparison of the two designs using results from synthesis is given in Section 5.

4 Decimal Floating-Point Multifunction Unit

There are several operations defined in IEEE P754 that can use hardware available in the decimal floating-point adder. In this section, we described how six other decimal operations can be integrated into the adder's datapath with only a small increase in area and delay.

4.1 SameQuantum, Quantize, and RoundToIntegral

SameQuantum and Quantize are the only two decimal-specific operations defined in IEEE P754. SameQuantum(A, B) compares the exponents of A and B and outputs true if they are the same and false if they are different. Since signaling and quiet NaNs are valid operands to SameQuantum, it does not signal any exceptions. SameQuantum is implemented by extending the 'Subtract and Absolute Value' (SUB-ABS) module in the 'Operand Alignment Calculation and Swapping Unit' (OACSU). The original SUB-ABS module computes $|EA_1 - EB_1|$ and outputs a $swap$ signal. To perform SameQuantum, logic is added to detect if $|EA_1 - EB_1|$ is zero.

Quantize(A, B) generates a decimal floating-point number that has the same value as A and the same exponent as B, unless rounding or an exception occurs. Due to the length of the significand in the destination format, Quantize sometimes raises an inexact or invalid operation flag. For example, if the exponent of B is larger than the exponent of A, the significand of A is right-shifted and rounding occurs based on the prevailing rounding mode. In this case, the inexact flag is raised if any nonzero digit is discarded. On the other hand, if the exponent of B is smaller, the significand of A is left-shifted and therefore it is possible that the required length of the significand is greater than the length of the destination format. In this case, the invalid operation flag is raised and the output is a quiet NaN. Thus, Quantize(A, B) is equivalent to rounding A only when $EA_1 < EB_1$.

The Quantize operation is implemented by modifying the OACSU to handle several special cases and performing decimal addition with CB_1 set to zero before sending it to OACSU. For example, if $EA_1 \geq EB_1$, CA_1 is left-shifted and the invalid operation flag is raised if the required length of the result is longer than the length of the destination format. Also, if $EA_1 < EB_1$, CA_1 needs to be right-shifted even though $CB_1 \equiv 0$. To provide the correct sign bit and rounding action for Quantize in this case, the EOP is forced to "ADD" even when the sign bit of A is negative.

RoundToIntegral(A) rounds a floating-point number to an integer based on the prevailing rounding mode. RoundToIntegral(A) is easily implemented as Quantize(A, 0) by setting CB_1 to zero and setting EB_1 to the bias of the exponent in the destination format. To avoid the condition where the invalid operation flag is raised and a quiet NaN is generated in Quantize, a 'Special Operation Unit' examines the exponent of A and selects A as the final result if $EA_1 \geq 0$.

4.2 Compare, minNum, and maxNum

Compare(A, B) compares A and B and indicates if $A > B$, $A < B$, $A \equiv B$, or A and B are unordered, which occurs if A or B is NaN. minNum(A, B) returns A if $A < B$ and returns B if $B < A$, while maxNum(A, B) returns A if $A > B$ and returns B if $B > A$. For both minNum and maxNum, if one operand is NaN and the other operand is a number, the operand that is a number is returned. As mentioned in Section 2, members of a cohort represent the same value. However, to ensure identical results on different platforms, they are not considered equal when performing minNum or maxNum. For example, $min(-1234 \times 10^5, -12340 \times 10^4) = -1234 \times 10^5$ and $max(-0, +0) = +0$. Unlike minNum and maxNum operations, the compare operation only compares the values of its input operands and therefore in the above examples, the two input operands are considered to be equal.

To implement Compare, minNum, and maxNum, our multifunction unit reuses the original decimal floating-point adder with the Operation set to Subtract. Since the significands are aligned, and the sign bit of the result and the relationship between the exponents of the operands are generated by the original design, all of the normal and the special cases mentioned above are implemented by adding a 'Spe-

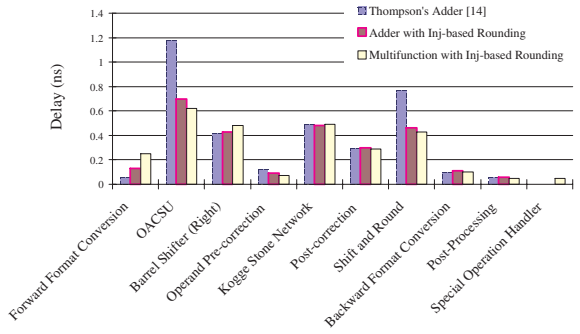

Figure 7. Delay Comparison

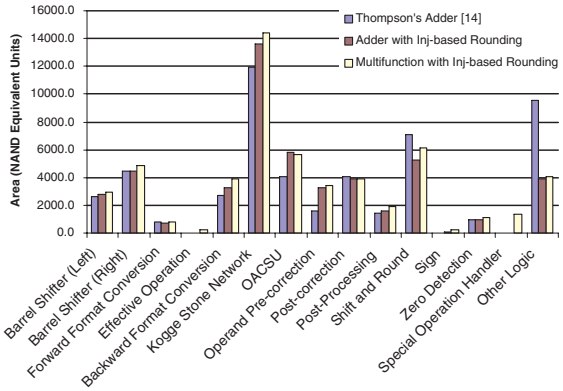

Figure 8. Area Comparison

cial Operation Unit' to the design. As most of the functions in this unit are performed in parallel with the original datapath, the only increase in the overall critical path delay is from a 64-bit 2-to-1 multiplexer.

5 Hardware Designs and Synthesis Results

Both decimal floating-point adders and the multifunction unit were modeled using RTL Verilog and then simulated using Modelsim and a comprehensive testbench generated using the decNumber library (version 3.32). Random, pattern-based, and corner-case testing were performed to ensure the correctness of the design. For a fair comparison, the adder design from [19] was extended to have the same functionality (i.e. handling both normal and special operands) as the proposed injection-based adder.

The decimal floating-point adders and multifunction unit were synthesized using Synopsys' Design Compiler and the 0.11um Gflx-p standard cell library from LSI Logic under normal operating condition. The clock, input signals, and output signals are assumed to be ideal. Inputs and outputs of the design are registered and the design is optimized for delay.

Figure 7 and Figure 8 compare the critical delay path and the area of the designs, respectively, when they are not pipelined. Table 4 compares the total area and delay of the three designs. As shown in Figure 7, the proposed injection-based adder reduces the latency in the 'Operand Alignment Calculation Unit' and the 'Shift and Round Unit' significantly, compared to the design presented in [19]. The proposed adder requires more area in the K-S network due to the generation of flags for the 'Post-correction Unit' and the trailing-nine detection, and in the 'Pre-correction Unit' due to the round-injection logic. However, the 'Shift and Round Unit' is smaller and less random logic exists in the proposed adder than in the design in [19].

From Table 4, our proposed decimal floating-point adder has about 21% less delay and 1.6% less area than the design presented in [19]. The proposed multifunction unit has 2.8% more delay and 9.7% more area than our decimal floating-point adder. Compared to the theoretical FO4 delay calculation for the double precision binary floating-point adder presented in [14], which uses a dual-path technique, our decimal injection-based adder has roughly 64% more delay.

Table 4. Delay and Area Comparison

	Adder from [19]	Injection-based Adder	Multifunction Unit
Delay (ns)	3.50	2.76	2.84
Delay (FO4)	63.6	50.2	51.6
Area (mm^2)	0.1451	0.1428	0.1566
Area (NAND Gates)	22443.3	22085.5	24232.9

To fit in a processor datapath, the designs should be pipelined to have roughly 10 to 18 FO4 inverter delays per stage. Based on the data from Table 4, a decimal floating-point adder or multifunction unit requires 3 or 4 pipeline

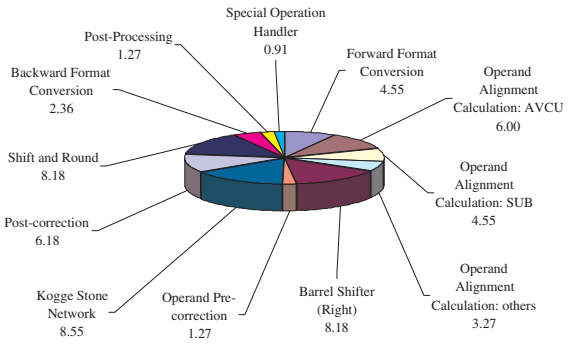

Figure 9. The Critical Path Delay in FO4 Inverter Delays of the Multifunction Unit

stages. To pipeline the multifunction unit, we can partition the design into 4 stages with a latency of about 15 FO4 inverter delays per stage (about 0.77ns per stage in Gflxp 0.11um technology). Figure 9 shows the delay of each module on the critical path. From the figure, to achieve the required delay, we can group the 'Forward Format Conversion', the 'Leading Zero Detectors', the 'Subtract and Absolute Value' (SUB-ABS) and the binary subtractor in the 'Operand Alignment Calculation and Swapping Unit' in one stage (14.91 FO4). The rest of the logic in the 'Operand Alignment Calculation and Swapping Unit', 'Barrel Shifters' and 'Operation Pre-correction Unit' can form the second stage (12.73 FO4). The 'K-S Network' and the 'Post-correction Unit' can be in the third stage (14.73 FO4), and all the remaining logic can be in the fourth stage (12.73 FO4). We can partition the pipeline of our decimal floating-point adder in a similar fashion.

6 Conclusion

In this paper, hardware implementations of a decimal floating-point adder and a multifunction unit were introduced. We described in detail several novel components in the designs, including those for the adder with injection-based rounding, extension to the decimal multifunction unit, and datapath parallelism. The proposed decimal adder inserts injection values during the operand pre-correction stage and applies injection correction values if needed, to accelerate the generation of results and the detection of the overflow flag. We extended the decimal floating-point adder to support eight operations with only a minor increase in delay and area. Finally, we have provided a detailed analysis on our synthesis results and a comparison between a previous design from [19] and the proposed decimal floating-point adder design and multifunction unit designs. Synthesis results show that the proposed adder design has 21% less delay and 1.6% less area than the previous design in [19] and the multifunction unit only has about 2.8% more delay and 9.7% more area than the proposed decimal floating-point adder.

Acknowledgement

This work was supported by the UW-Madison Graduate School and an IBM Faculty Award. The authors would like to thank Ying-Cherng Lan and Professor Sao-Jie Chen from National Taiwan University for the help they provided with design experience.

References

[1] 754 Working Group. Draft of the IEEE standard for floating-point arithmetic, November 2006.

[2] A. Beaumont-Smith and C.-C. Lim. Parallel prefix adder design. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 218–225, 2001.

[3] G. Bohlender and T. Teufel. BAP-SC: A decimal floating-point processors for optimal arithmetic. In *Computer arithmetic: Scientific Computation and Programming Languages*, pages 31–58. B.G Teubner Verlag, 1987.

[4] W. Bultmann, W. Haller, H. Wetter, and A. Worner. Binary and decimal adder unit. *U.S. Patent 6,292,819*, September 2001.

[5] N. Burgess. Prenormalization rounding in IEEE floating-point operations using a flagged prefix adder. *IEEE Transactions on VLSI System*, 13(2):266–277, Feb 2005.

[6] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough. The IBM z900 decimal arithmetic unit. In *Proceedings of the 35th Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1335 –1339. IEEE Computer Society, November 2001.

[7] M. S. Cohen, T. E. Hull, and V. C. Hamacher. CADAC: A controlled-precision decimal arithmetic unit. *IEEE Transactions on Computers*, 32(4):370–377, 1983.

[8] M. F. Cowlishaw. Densely packed decimal encoding. In *IEE Proceedings - Computers and Digital Techniques*, volume 149, pages 102–104, May 2002.

[9] G. Even and P. M. Seidel. A comparison of three rounding algorithms for IEEE floating-point multiplication. *IEEE Transactions on Computers*, 49(7), July 2000.

[10] W. Haller, U. Krauch, T. Ludwig, and H. Wetter. Combined binary/decimal adder unit. *U.S. Patent 5,928,319*, July 1999.

[11] H. Nikmehr, B. Phillips, and C. C. Lim. A decimal carry-free adder. In *Proceedings of SPIE, SPIE Symposium on Smart Materials, Nano-, and Micro-Smart Systems*, volume 5649, pages 786–797, February 28, 2005 2005.

[12] R. Sacks-Davis. Applications of redundant number representations to decimal arithmetic. *The Computer Journal*, 25(4):471–477, 1982.

[13] M. S. Schmookler and A. W. Weinberger. High speed decimal addition. *IEEE Transactions on Computers*, C-20:862–867, Aug 1971.

[14] P. M. Seidel and G. Even. Delay-optimized implementation of IEEE floating-point addition. *IEEE Transactions on Computers*, 53(2):97–113, 2004.

[15] S. Shankland. IBM's POWER6 gets help with math, multimedia. *ZDNet News*, October 2006.

[16] B. Shirazi, D. Y. Y. Yun, and C. N. Zhang. RBCD: redundant binary coded decimal adder. *IEE Proceedings on Computers and Digital Techniques*, 136(2):156–160, 1989.

[17] Sun Microsystem. BigDecimal class, Java 2 platform standard edition 5.0, API specification, 2004.

[18] A. Svoboda. Decimal adder with signed digit arithmetic. *IEEE Transactions on Computers*, C-18(3):212–215, 1969.

[19] J. Thompson, M. J. Schulte, and N. Karra. A 64-bit decimal floating-point adder. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI, Lafayette, LA,*, pages 297–298, Feb 2004.