Hardware Designs for Decimal Floating-Point Addition and Related Operations

Liang-Kai Wang, *Member*, *IEEE*, Michael J. Schulte, *Senior Member*, *IEEE*, John D. Thompson, and Nandini Jairam

Abstract—Decimal arithmetic is often used in commercial, financial, and Internet-based applications. Due to the growing importance of decimal floating-point (DFP) arithmetic, the IEEE 754-2008 Standard for Floating-Point Arithmetic (IEEE 754-2008) includes specifications for DFP arithmetic. IBM recently announced adding DFP instructions to their POWER6, z9, and z10 microprocessor architectures. As processor support for DFP arithmetic emerges, it is important to investigate efficient arithmetic algorithms and hardware designs for common DFP arithmetic operations. This paper gives an overview of DFP arithmetic in IEEE 754-2008 and discusses previous research on decimal fixed-point and floating-point addition. It also presents novel designs for a DFP adder and a DFP multifunction unit (DFP MFU) that comply with IEEE 754-2008. To reduce their delay, the DFP adder and MFU 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 DFP adder is roughly 21 percent faster and 1.6 percent smaller than a previous DFP adder design, when implemented in the same technology. Compared to the DFP adder, the DFP MFU provides six additional operations, yet only has 2.8 percent more delay and 9.7 percent more area. A pipelined version of the DFP MFU has a latency of six cycles, a throughput of one result per cycle, an estimated critical path delay of 12.9 fanout-of-four (FO4) inverter delays, and an estimated area of 45,681 NAND2 equivalent gates.

Index Terms—Decimal, floating-point, computer arithmetic, addition, subtraction, multifunction unit, logic design.

1 Introduction

Binary floating-point (BFP) arithmetic is usually sufficient for scientific applications. However, it is not acceptable for many commercial and financial applications. Decimal numbers in these applications are usually required to be represented exactly, and arithmetic operations often need to mirror manual decimal calculations, which perform decimal rounding. Therefore, these applications often use software to perform decimal arithmetic operations. Although this approach eliminates representation errors 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 (DFP) arithmetic, specifications for it have been added to the IEEE 754-2008 Standard for Floating-Point Arithmetic (IEEE 754-2008) [1]. Recently, IBM announced adding DFP instructions to their POWER6, z9, and z10 microprocessor architectures [2], [3], [4]. These DFP instructions produce results that are compliant with IEEE 754-2008.

- *L.-K. Wang is with Advanced Micro Devices (AMD) Long Star Design Center. 7171 Southwest Parkway, Suite B400.621, Austin, TX 78735. E-mail: liang-kai.wang@amd.com.*
- *M.J. Schulte is with the University of Wisconsin-Madison, 1415 Engineering Dr., Madison, WI 53706. E-mail: schulte@engr.wisc.edu.*
- *J.D. Thompson is with Cray Inc., 1050 Lowater Road, PO Box 6000, Chippewa Falls, WI 54729. E-mail: johnt@cray.com.*
- *N. Jairam is with Intel Corp., 1900 Prairie City Road, Folsom, CA 95630. E-mail: nandini.jairam@intel.com.*

Manuscript received 31 July 2007; revised 30 Mar. 2008; accepted 16 June 2008; published online 6 Aug. 2008.
Recommended for acceptance by J.-C. Bajard.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-07-0397.
Digital Object Identifier no. 10.1109/TC.2008.147.

In this paper, we present a DFP adder that uses a parallel method for decimal operand alignment, and a modified Kogge-Stone (K-S) parallel prefix network [5] for significand addition and subtraction. It also applies novel decimal variations of the injection-based rounding method [6] and the flagged prefix network [7] to decrease the latency of rounding and overflow detection. The DFP adder supports all the rounding modes and appropriate exceptions specified in IEEE 754-2008 and all the rounding modes specified in the Java BigDecimal library [8]. It has 21 percent less delay and 1.6 percent less area than the DFP adder presented in [9], when implemented in the same technology. The DFP adder design is extended to implement a DFP multifunction unit (DFP MFU) that performs eight DFP operations defined in IEEE 754-2008: addition, subtraction, compare, minNum, maxNum, quantize, sameQuantum, and roundToIntegral. Synthesis results show that our DFP MFU has only 2.8 percent more delay and 9.7 percent more area than our DFP adder. The DFP adder and MFU presented in this paper support 64-bit DFP operands, but the techniques presented in this paper can be extended to handle other operand sizes and other DFP operations.

The rest of this paper is organized as follows: Section 2 gives an overview of DFP arithmetic in IEEE 754-2008. Section 3 presents related research on decimal addition. Section 4 describes our proposed DFP adder with injection-based rounding. Section 5 discusses the DFP MFU. Section 6 presents synthesis results for our DFP adder and MFU and for the DFP adder from [9]. Section 7 discusses optimizations that can be made to our DFP adder and MFU designs to potentially speedup common cases in real applications. Section 8 gives our conclusions. This paper is an extension

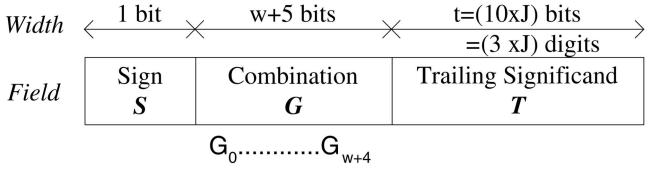


Fig. 1. Decimal interchange floating-point format.

of the research presented in [10] and summarizes research presented in [9].

In this paper, SX_Y, CX_Y, and EX_Y are the sign, significand, and exponent of a DFP number, respectively. X is A, B, or R to denote operands or result, respectively. The subscript "$_Y$" is a digit that denotes the output of different modules. $(N)_i^j$ refers to the jth bit in digit position, i, in a number, N, where the least significant bit (LSB) and the least significant digit (LSD) have index 0. For example, $(CA_1)_2^3$ is bit three of digit two in the significand CA_1.

2 Decimal Arithmetic in IEEE 754-2008

2.1 Decimal Floating-Point Formats

IEEE 754-2008, which was officially approved in June 2008, is the revised version of the IEEE 754 Standard for BFP arithmetic, which was originally ratified in 1985 [11]. IEEE 754-2008 defines decimal interchange formats that are used for storing data and exchanging data between platforms. These formats are designed for storage efficiency and numbers in these formats are converted to an internal format before they are processed. IEEE 754-2008 defines a 32-bit storage format called decimal32, and 64-bit and 128-bit basic formats called decimal64 and decimal128, respectively. The decimal64 and decimal128 formats are used for both storage and computations.

In IEEE 754-2008, the value of a finite DFP number with an integer significand is

$$v = (-1)^S \times C \times 10^q, \tag{1}$$

where S is the sign, q is the unbiased exponent, and C is the significand, which is a nonnegative integer of the form $c_0c_1c_2 \ldots c_{p-1}$ with $0 \le c_i < 10$. p is the precision or the length of the significand, which is equal to 7, 16, or 34 digits, for decimal32, decimal64, or decimal128, respectively.

The IEEE 754-2008 decimal interchange format is shown in Fig. 1. The 1-bit Sign Field S indicates the sign of a number. The $(w+5)$-bit Combination Field G provides the most significand digit (MSD) of the significand and a nonnegative biased exponent E such that $E = q + bias$. The G Field also indicates special values, such as Not-a-Number (NaN) and infinity (∞). The remaining digits of the significand are specified in the t-bit Trailing Significand Field T. IEEE 754-2008 specifies two encodings for the Trailing Significand Field. The first encodes its significand using a decimal encoding, also known as the Densely Packed Decimal (DPD) encoding. The other encoding uses a binary integer significand, and is commonly referred to as the Binary Integer Decimal (BID) encoding. IEEE 754-2008 refers to the BID encoding as the binary encoding of DFP numbers and it refers to the DPD encoding as the decimal encoding of DFP numbers. More details about the DPD and BID encodings are given in IEEE 754-2008 [1]. Table 1 gives the important parameters used in the standard for each decimal format. In this table, widths are given in bits, and $emax$ and $emin$ indicate the minimum and maximum unbiased exponents, respectively, in each format.

TABLE 1
Decimal Interchange Format Parameters

Format Name	decimal32	decimal64	decimal128
Storage width	32	64	128
Trailing significand field width (t)	20	50	110
Combination field width ($w+5$)	11	13	17
Number of significand digits (p)	7 digits	16 digits	34 digits
Exponent bias	101	398	6176
$emax$	+96	+384	+6144
$emin$	-95	-383	-6143

2.2 Rounding Modes and Decimal-Specific Operations

IEEE 754-2008 specifies five rounding modes: roundTiesToEven rounds the result to the nearest representable floating-point number and selects the number with an even LSD if a tie occurs; roundTiesToAway rounds the result to the nearest representable floating-point number and selects the number with the larger magnitude if a tie occurs (roundTiesToAway is a required rounding mode for DFP arithmetic, but not for BFP arithmetic); roundTowardPositive rounds the result toward positive infinity; roundTowardNegative rounds the result toward negative infinity; and roundTowardZero truncates the result.

Financial applications tend to use symbols to define units. For example, "K" for thousand, "M" for million, "B" for billion, and "%" for hundredths. Some database systems store values using these symbols, instead of the IEEE 754-2008 formats. Therefore, numbers are aligned to these symbols before the significands of the numbers are extracted to be stored in databases. On the other hand, programs may need to compare values in one database against the other to determine if they are in the same unit (i.e., quantum) before further computation. To simplify conversions and comparisons, IEEE 754-2008 defines two decimal-specific operations: SameQuantum and Quantize. More details on these two operations are given in Section 5.

2.3 Characteristics of Decimal Numbers and Exceptions

As described in IEEE 754-2008, the significand of a DFP number is not normalized, which means that a single DFP number may have multiple representations. A website developed by Mike Cowlishaw gives some examples explaining why decimal numbers should not be normalized [12]. A set of these equivalent decimal numbers is called the cohort of a DFP number. Because of this characteristic, IEEE 754-2008 defines the term, preferred exponent, which specifies the exponent, and implicitly the significand, after each DFP operation. For the DFP addition, $x + y$, if the result

TABLE 2
Prepared Exponents for Operations in DFP MFU

Decimal Operation	Preferred exponent for exact result	Preferred exponent for inexact result
addition $x + y$	$min(Q(x), Q(y))$	Least possible exponent
subtraction $x - y$	$min(Q(x), Q(y))$	Least possible exponent
quantize(x, y)	$Q(y)$	
minNum(x, y)	$Q(x)$ if x is the result and $Q(y)$ if y is the result.	
maxNum(x, y)	$Q(x)$ if x is the result and $Q(y)$ if y is the result.	
roundToIntegral(x)	$max(Q(x), 0)$	

cannot be represented exactly in the destination DFP format, the preferred exponent is the least possible exponent of the result, so as to preserve the maximum precision in the significand. For example, if $x = 400 \times 10^2$, $y = 105 \times 10^{-3}$, and the destination DFP format is decimal32 with $p = 7$, then $x + y = 4{,}000{,}011 \times 10^{-2}$ with roundTiesToAway. If the result after DFP addition can be represented exactly in the destination DFP format, the preferred exponent of the result is $min(Q(x), Q(y))$, where $Q(x)$ and $Q(y)$ are the exponents of x and y, respectively. For example, if $x = 400 \times 10^2$, $y = 105 \times 10^{-2}$, and the destination DFP format is decimal32, then $x + y = 4{,}000{,}105 \times 10^{-2}$. Table 2 shows the preferred exponents after decimal operations in our DFP MFU. More details on the preferred exponent are given in IEEE 754-2008 [1].

There are a few exceptions that need to be handled by our DFP MFU. These include Inexact, Invalid Operation, and Overflow. Underflow is not possible for any operation in the DFP MFU because all the operations in this unit only generate either inexact or subnormal results, but not both. Table 3 shows the conditions for each exception and the corresponding output. In this table, MAXFLOAT is the largest DFP number in the destination format.

3 Related Research

Previous research on decimal addition and subtraction has focused on fixed-point operations. Decimal numbers are often represented in binary coded decimal (BCD). Unlike binary addition, for which carry generation is simple, BCD addition requires carry computations across digit boundaries, as 6 out of the 16 combinations in a BCD digit are not used. To generate correct carry and sum digits, those unused combinations (1010_2 to 1111_2) need to be skipped.

TABLE 3
Exceptions for Operations in DFP MFU

Exception	Condition	Output
Inexact	The exact result cannot be represented in the destination format	Rounded result
Invalid Operation	· Either operand is a sNaN · ∞ - ∞ · Insufficient space for Quantize	Quiet NaN
Overflow	The result exceeds the largest finite number in the destination format	$\pm\infty$ or $\pm$MAXFLOAT

BCD Addition	CA = 0011 0010 1001	329
	+ CB = 1001 0000 1001	+ 909
Add Pre-Correction	CA = 0011 0010 1001	
	+ P = 0110 0110 0110	
Add the Addend	PA = 1001 1000 1111	
	+ CB = 1001 0000 1001	
Generate Sum and Carry	S = 1 0010 1001 1000	
	C = 1 0 1	
Subtract Post-Correction	S = 1 0010 1001 1000	
	− P' = 0000 0110 0000	
	POS = 1 0010 0011 1000	1238

Fig. 2. An example of BCD addition.

This is often done by adding six (0110_2) to each BCD digit. If a digit carry-out does not occur, the bias of six is subtracted from that digit position [13], [14], [15], [16]. With BCD subtraction, bits in the subtrahend are inverted. The result is corrected after the subtraction based on the sign of the result and the carry-out of each digit.

An example of BCD addition is shown in Fig. 2, where a precorrection value P is added to the augend CA to obtain an intermediate result PA. The addend CB is added to PA to obtain a temporary sum S and digit carry vector C, which determines if a postcorrection value P' should be used to adjust each sum digit. In this example, only the second digit (i.e., $(S)_1$) needs to be corrected because its carry-out is zero. Therefore, six is subtracted from $(S)_1$ to form the final result POS.

Unlike traditional BCD addition, which uses precorrection and postcorrection, Schmookler and Weinberger present 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 the final sum digits quickly [17]. In Schmookler's design, $(G)_i^j = (A)_i^j \wedge (B)_i^j$ and $(P)_i^j = (A)_i^j \vee (B)_i^j$ are bit generate and propagate signals for digit i, respectively, where $\wedge$ denotes logical AND and $\vee$ denotes logical OR. Based on these two sets of variables, for each digit at position i, two signals K_i and L_i are produced, where

$$\begin{aligned} K_i \equiv \left\{sum_i^{3:1} \geq 10\right\} &= (G)_i^3 \vee \left((P)_i^3 \wedge (P)_i^2\right) \vee \left((P)_i^3 \wedge (P)_i^1\right) \\ &\quad \vee \left((G)_i^2 \wedge (P)_i^1\right), \\ L_i \equiv \left\{sum_i^{3:1} \geq 8\right\} &= (P)_i^3 \vee (G)_i^2 \vee \left((P)_i^2 \wedge (G)_i^1\right). \end{aligned} \tag{2}$$

K_i is a digit generate signal, L_i is a digit propagate signal, and $sum_i^{3:1}$ is the binary value of the digit sum of $(A)_i + (B)_i$ when its LSB is not included. The carry-out of each digit is defined as

$$Cout_i = K_i \vee \left(L_i \wedge (C)_i^1\right), \tag{3}$$

where $(C)_i^1$ is the carry-out of the least significant sum bit and has a weight of 2. The digit carry-propagate network uses a binary parallel-prefix tree, and the sum digits are computed using $(A)_i$'s, $(B)_i$'s, and $(C)_i^1$'s. Schmookler's addition scheme is faster than the normal precorrection and postcorrection method when only performing BCD

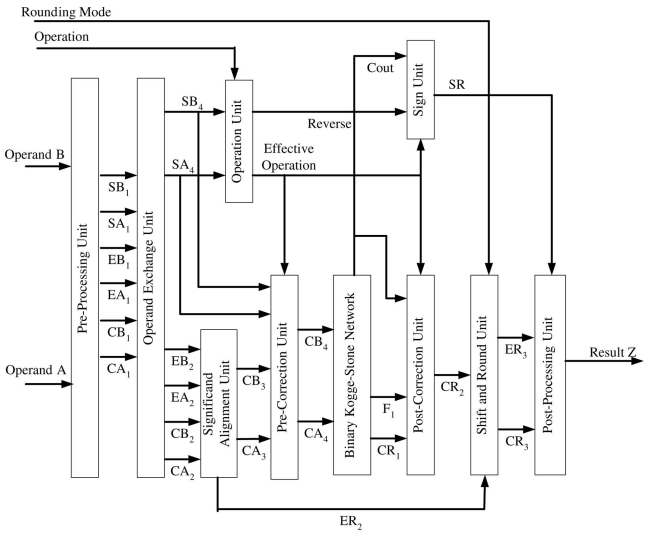


Fig. 3. Thompson's DFP adder [9].

addition. For BCD subtraction, nine's complement logic is needed before and after the adder to generate correct results. This approach is used in the IBM S/390 machines.

Details on other techniques for decimal fixed-point addition, including decimal signed-digit addition and decimal multioperand addition, are summarized in [18].

In [9], Thompson et al. implement the first IEEE 754-2008 compliant DFP adder. The block diagram of their design is shown in Fig. 3. In their adder, the "Preprocessing Unit" is used to extract significands, sign bits, and exponents from both operands. Next, the "Operand Exchange Unit" and "Significand Alignment Unit" perform operand swapping and alignment based on the exponents. In parallel, the "Operation Unit" generates the effective operation EOP based on the sign bits of the input operands and the $Operation$ signal. The outputs from the "Significand Alignment Unit" enter the "Precorrection Unit," which uses a modified excess-3 decimal encoding as the internal encoding to realize an overall bias of six for both addition and subtraction. This unit also inverts the excess-3 encoded subtrahend if the effective operation is subtraction and expands the sticky bit to a sticky digit. It simplifies the design to perform the excess-3 encoding and subtrahend inversion after the operands have been swapped, the alignment shift is performed, and the effective operation is determined. The excess-3 encoded operands then enter the "Binary K-S Network" to produce a computed sum vector CR_1 and a flag vector F_1, which is used to adjust the result when it is positive and EOP is subtraction. The "Postcorrection Unit" adjusts the result based on the sign of the result EOP, the carry vector, and the flag vector. The corrected result CR_2 enters the "Shift and Round Unit," which performs shifting, rounding, and overflow detection if needed. Finally, the "Postprocessing Unit" combines the sign bit, the significand, and the exponent to form an IEEE 754-2008 result in the decimal64 DPD format. This unit also changes the result to special values, such as NaN, $\pm\infty$, or $\pm$MAXFLOAT, which is the maximum representable DFP number, based on the prevailing rounding mode, the overflow flag, the sign of the result, and if either of the input operands is a special operand.

In the IBM System z9 microprocessor, DFP addition and subtraction are performed through a combination of dedicated hardware and millicode, which is the lowest layer of firmware in this architecture [3]. To perform DFP addition, the processor

1. reads operands in the IEEE 754-2008 DPD format from a Floating-Point Register (FPR) into a Millicode General-purpose Register (MGR),
2. extracts the signs, significands, and exponents and stores them into MGRs,
3. performs operand alignment, decimal fixed-point addition, and rounding,
4. determines the result's sign and exponent,
5. compresses the sign, significand, and exponent to form an IEEE 754-2008 DPD result, and
6. stores the result in a FPR.

The System z9 mainframe uses millicode operations to implement DFP instructions since this allows it to take advantage of existing decimal fixed-point hardware and provides flexibility for future optimizations. Simulation results from [3] show that DFP addition and subtraction operations take between 100 and 150 cycles in the IBM z9 microprocessor.

The IBM POWER6 microprocessor implements several DFP operations, including addition and subtraction, with a 36-digit decimal adder [2]. The decimal adder is composed of several 4-digit decimal conditional adders and is capable of performing decimal operations on both doubleword (16-digit) and quadword (34-digit) operands. The 36-digit adder is split into two parts, each of which is 18 digits wide to allow for 16 digits of precision, a guard digit, and a round digit for doubleword operations or 34 digits of precision, a guard digit, and a round digit for quadword operations. The adder can perform two simultaneous doubleword operations or one quadword operation. DFP addition and subtraction require preprocessing, rounding, and postprocessing to ensure their results are compliant with IEEE 754-2008. The latency of DFP addition in the POWER6 processor varies based on the operands. In the worst case scenario, operands need to be converted from the IEEE 754-2008 format to the BCD format, swapped if needed, left shifted, right shifted, and right shifted a second time, before the two aligned operands are added. After the addition, the result is rounded and compressed to the IEEE 754-2008 format. The worst case latency for DFP addition with decimal64 operands is 17 cycles and the cycle time is equivalent to roughly 13 FO4 inverter delays.

4 Decimal Floating-Point Adder

4.1 Overview of the Decimal Floating-Point Adder

Our DFP adder is based on Thompson et al.'s adder [9], but it includes significant enhancements and modifications to reduce delay. Fig. 4 shows a high-level block diagram of our DFP adder. The "Forward Format Conversion Unit" takes two IEEE 754-2008-encoded operands A and B and the $Operation$, and produces sign bits SA_1 and SB_1, BCD significands CA_1 and CB_1, biased exponents EA_1 and EB_1, and the effective operation EOP (not shown in the figure). The "Operand Alignment Calculation and Swapping Unit"

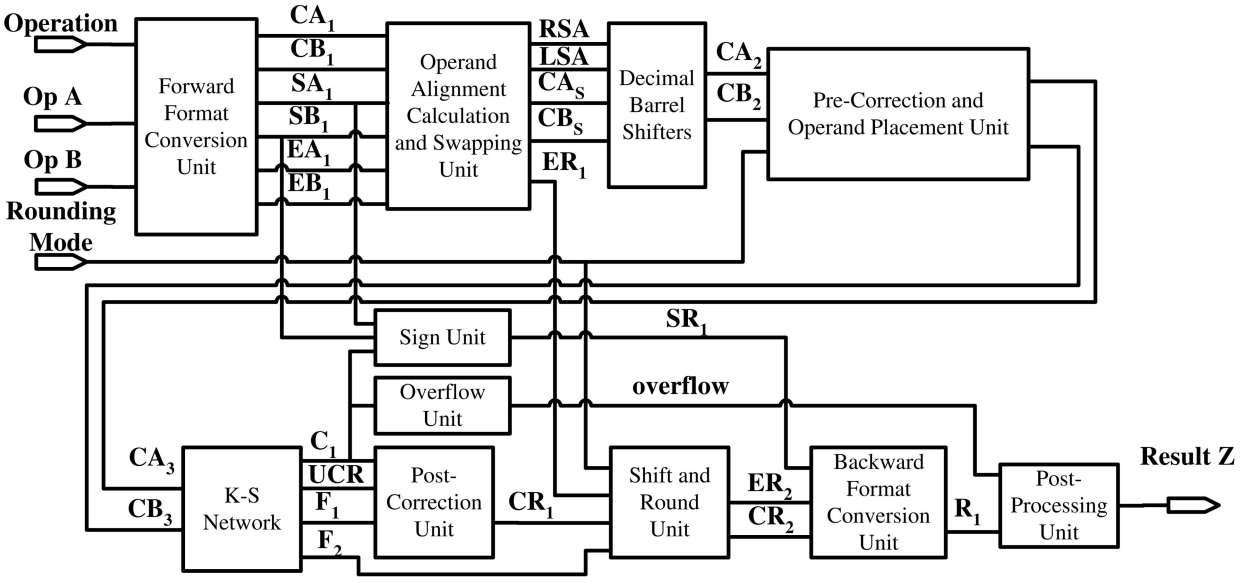


Fig. 4. Proposed DFP adder.

(OACSU) takes these values and computes the result's temporary exponent ER_1, right shift amount RSA, and 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 and Operand Placement Unit." Based on the EOP signal and the prevailing rounding mode, the "Precorrection and Operand Placement Unit" prepares the BCD operands for addition or subtraction and injects a value needed for rounding.

The corrected significands CA_3 and CB_3 are then fed into the "K-S Network" [5], which produces an uncorrected result UCR, a digit-carry vector C_1, and flag vectors F_1 and F_2. After this, the "Postcorrection 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. In parallel, the "Sign Unit" and "Overflow Unit" compute the result's sign bit SR_1 and the overflow signal. The result values CR_2, ER_2, and SR_1 are combined to generate an IEEE 754-2008 DPD-encoded result in the "Backward Format Conversion Unit." This result and the original input operands are examined in the "Postprocessing Unit" to determine if a special result is needed, which happens if either one or both of the input operands are NaN or $\pm\infty$. Based on the overflow flag, the sign of the result, and the prevailing rounding mode, this unit may also set the result to $\pm\infty$ or $\pm$MAXFLOAT. Further details on each of these units and an example of DFP subtraction are provided in the following sections.

4.2 Forward Format Conversion and Operand Alignment Calculation and Swapping

The core of the DFP adder operates on BCD significands. Therefore, converters are first employed to extract the BCD-encoded significands, binary exponents, and sign bits from both IEEE 754-2008-encoded operands. The two DPD-encoded significands are simultaneously converted to BCD-encoded significands. 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. 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, 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, and $\oplus$ denotes exclusive-OR.

Decimal operand alignment is more complex than its binary counterpart because decimal numbers are not normalized. This leads to the potential for 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:

$$\begin{aligned} LSA &= min(|EA_1 - EB_1|, LA_S), \\ RSA &= min(max(|EA_1 - EB_1| - LA_S, 0), p+3), \\ ER_1 &= EA_S - LSA, \end{aligned} \tag{4}$$

where p is the precision of the DFP format. 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 at most LA_S digits so that the left-shifted significand CA_2 does not have more than p digits, where p is equal to 16 in the decimal64 format. 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 operand alignment. RSA is equal to zero if LA_S is large enough to accommodate the exponents' difference. RSA is also limited to at most $p+3$ digits, since the right-shifted significand CB_2 contains p digits plus guard, round, and sticky digits, as explained in Section 4.3. The temporary result exponent ER_1 is simply

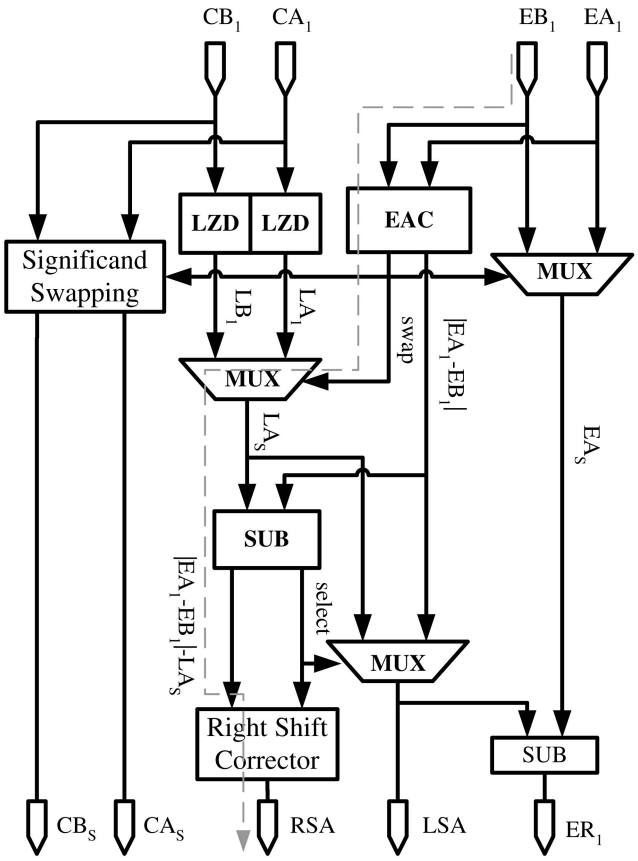


Fig. 5. OACSU.

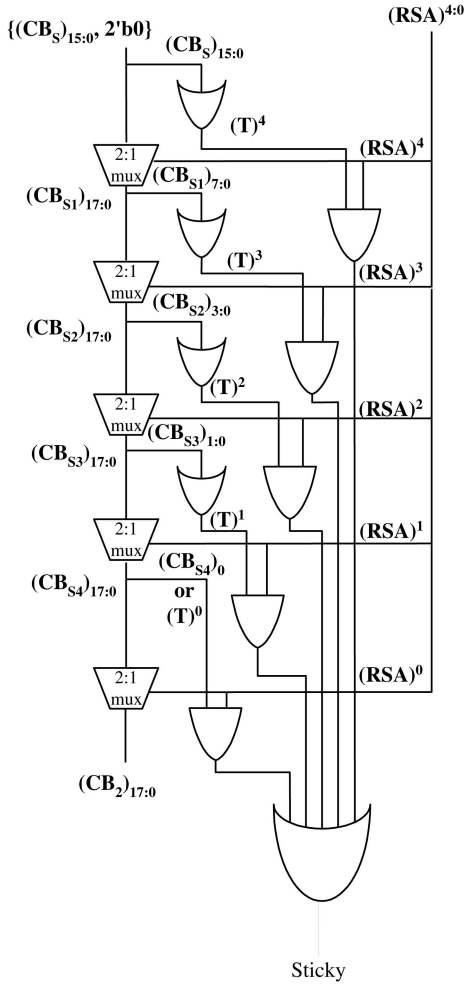


Fig. 6. Decimal barrel shifter and sticky digit generation.

the larger exponent EA_S after it has been adjusted to compensate for the left shift amount LSA.

The technique used in [9] to perform operand swapping 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 and is on the critical delay path.

To reduce the delay, our design uses an End Around Carry (EAC) adder [7] to compute $|EA_1 - EB_1|$ and $swap = sign(EA_1 - EB_1)$. In parallel, it performs leading zero detection on both CA_1 and CB_1 to produce LA_1 and LB_1. If $swap$ is one, then $CA_S = CB_1$, $CB_S = CA_1$, $LA_S = LB_1$, and $EA_S = EB_1$. Otherwise, $CA_S = CA_1$, $CB_S = CB_1$, $LA_S = LA_1$, 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. This approach is shown in Fig. 5, where the dashed line indicates the critical delay path of this unit. In this figure, RSA is limited to a value between 0 and $p + 3$ by the Right Shift Corrector and in parallel $ER_1 = EA_S - LSA$ is computed. Synthesis results indicate that our approach reduces the critical path delay of the "OACSU" by roughly 41 percent and increases its area by roughly 4.8 percent compared to the design in [9].

4.3 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 digit S. Fig. 6 illustrates how CB_S is shifted and how a sticky bit is generated from RSA and CB_S. The sticky bit is later expanded into a sticky digit in the "Precorrection and Operand Placement Unit" to allow all digits in CB_2 to be processed using the same technique and to simplify further processing. In Fig. 6, a series of multiplexers right shift CB_S based on the bits of RSA. In parallel with this, bits from CB_S or shifted values of CB_S from the multiplexer outputs are ORed to form the bits $(T)^{4:0}$. The bits of RSA, $(RSA)^i$, are used as mask bits to determine if $(T)^i$ should contribute to the sticky bit. The outputs from ANDing $(T)^i$ and $(RSA)^i$ are then ORed to form the sticky bit. Although Fig. 6 shows one method for generating the sticky bit, various optimization can be made based on the timing requirement of the overall design.

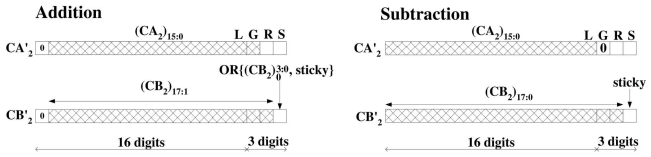


Fig. 7. Operand placement for DFP addition and subtraction.

For example, 4-to-1 multiplexers may be used, instead of 2-to-1 multiplexers, to reduce delay. With DFP arithmetic in IEEE 754-2008, 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 for DFP addition and subtraction.

Once shifted, a value based on a sign bit and prevailing rounding mode is injected into the R and S digit positions of CA_2 to form CA_2', which is a 19-digit BCD number, as shown in Fig. 7. The injection value, shown in Table 4, is determined by equations similar to those developed for BFP addition [19] and is used to facilitate correct rounding. The injection values are chosen such that including the injection value as part of the addition or subtraction effectively allows rounding to be replaced by truncation. For example, if the rounding mode is roundAwayZero, the injection value of $(R, S) = (9, 9)$ is used so that a carry is generated into the G digit position unless both the R and S digits of CB_2 are zero. To perform correct rounding in the roundTiesToEven rounding mode, the LSB of the result is set to zero in the Shift and Round Unit when the result is halfway between two representable DFP numbers (i.e., when $\mathrm{RS} = 00$ after the final addition).

In Table 4, roundTiesToZero and roundAwayZero are rounding modes used in the Java BigDecimal class, and all the others are required in IEEE 754-2008. $Sign_{inj}$ 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, LSA could be nonzero but RSA is always zero. Therefore, rounding is not needed. The sign bit used to select the injection value is computed as

$$\begin{aligned} Sign_{inj} = &\left(\left(\overline{EOP \wedge swap}\right) \wedge SA_1\right) \\ &\vee \left((EOP \wedge swap) \wedge (Operation \oplus SB_1)\right). \end{aligned} \tag{5}$$

Some rounding modes do not depend on the $Sign_{inj}$ bit to determine the injection values and this is denoted using "?" in the table.

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 Fig. 7, both operands are placed starting from one digit to the right of the MSD for addition and from the MSD for subtraction. This placement allows the 16-digit final result to always be selected from the 17 more significant digits and allows the injection correction value to be placed in the same locations for both effective addition and subtraction. The operands after placement are denoted as CA_2' and CB_2', and both are 19 digits. The injection value is inserted on all addition/subtraction-related operations, except when 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 postcorrection logic. To avoid this condition, another signal $flushing$ is generated to clear the injection value. This signal is computed as $flushing = EOP \wedge (RSA \equiv 0)$.

TABLE 4
Injection Values for Different Rounding Modes

$Sign_{inj}$	Rounding Modes	Injection Value (R, S)
?	roundTowardZero	(0 , 0)
?	roundTiesToAway	(5 , 0)
?	roundTiesToZero	(4 , 9)
?	roundTiesToEven	(5 , 0)
-	roundTowardPositive	(0 , 0)
+	roundTowardPositive	(9 , 9)
-	roundTowardNegative	(9 , 9)
+	roundTowardNegative	(0 , 0)
?	roundAwayZero	(9 , 9)

After the injection value is inserted into the data path, both operands are adjusted in order to generate correct carry-out digits. The equations implemented by the "Precorrection and Operand Placement Unit" are

$$\begin{aligned} (CA_3)_i &= \begin{cases} (CA_2')_i + 6, & \text{if } EOP \text{ is add,} \\ (CA_2')_i, & \text{otherwise,} \end{cases} \\ (CB_3)_i &= \begin{cases} (CB_2')_i, & \text{if } EOP \text{ is add,} \\ \overline{(CB_2')_i}, & \text{otherwise,} \end{cases} \\ & i = 0 \ldots 18, \end{aligned} \tag{6}$$

where $\overline{(CB_2')_i}$ is the fifteen's complement of $(CB_2')_i$ and is obtained by inverting each bit of $(CB_2')_i$.

With effective addition, each digit $(CA_2')_i$ is incremented by six such that in each digit position the operation performed is $\{(C_1)_{i+1}, (UCR)_i\} = ((CA_2')_i + 6) + (CB_2')_i + (C_1)_i$, where $(C_1)_i$ is the carry into digit i, $(UCR)_i$ is the uncorrected 4-bit result in digit position i, $\{(C_1)_{i+1}, (UCR)_i\}$ denotes the concatenation of $(C_1)_{i+1}$ and $(UCR)_i$, and $\{(C_1)_{i+1}, (UCR)_i\}$ is in the range of [6, 25]. With effective subtraction, the operation performed at each digit is $\{(C_1)_{i+1}, (UCR)_i\} = (CA_2')_i + (15 - (CB_2')_i) + (C_1)_i = (CA_2')_i + 6 + (9 - (CB_2')_i) + (C_1)_i$, and $\{(C_1)_{i+1}, (UCR)_i\}$ is in the range of [6, 25]. Having $\{(C_1)_{i+1}, (UCR)_i\}$ in the range [6, 25] helps generate correct carries using the K-S network because a correct carry is automatically generated into the next digit when $\{(C_1)_{i+1}, (UCR)_i\}$ is greater than 15. It also 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 4.5.

4.4 Kogge-Stone Network

Because both operands are adjusted, a binary K-S network [5] can be used to generate carries into each digit. In addition to the flag bits used in the postcorrection step (i.e., F_1 in this paper) [9], another set of flag bits F_2 is generated and used in the "Shift and Round Unit." The F_2 flag bits are used to avoid another carry-propagate addition when the MSD of CR_1 is nonzero. For example with $p = 7$, if $CA_3 = 0\_9999999\_99$, $CB_3 = 0\_0039999\_91$, and decimal addition with roundTowardPositive is performed, then CR_1 becomes

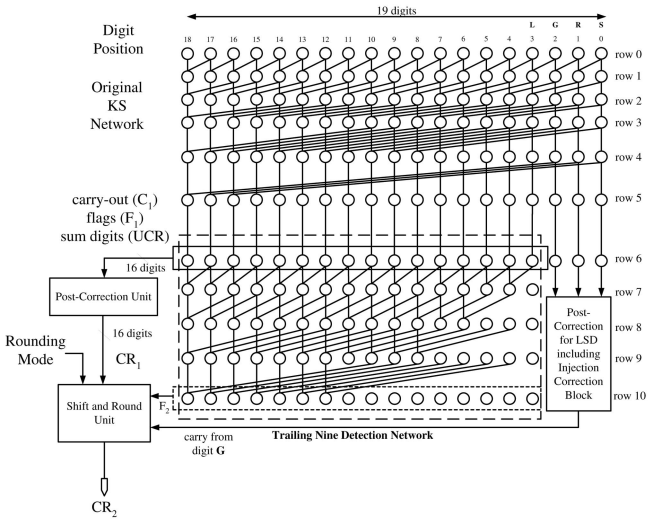


Fig. 8. K-S network and flag logic.

1_0039999_90 and has an MSD of 1. Examining the result indicates that there are three consecutive nines starting from the LSD (the two rightmost nines are discarded when $p = 7$). Therefore, the four LSDs are incremented and the final result becomes $1{,}004{,}000 \times 10^1$ after shifting and rounding. Determining which digits need to be incremented is performed by a method known as *trailing-nine detection*. It is important to note that trailing-nine detection is only used if EOP is add or EOP is 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 results is guaranteed to be less than 17 digits.

Fig. 8 illustrates how the original K-S network is extended to detect trailing nines. The traditional binary injection-based rounding method uses a compound adder 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 a decimal variation of the flagged-prefix method [7] to compute the uncorrected sum and the uncorrected sum plus one. Since the value generated in the K-S network is not in the BCD encoding, 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 MSDs.

An example of DFP subtraction is shown in Fig. 9, where F_1 is a flag vector that indicates the end of a continuous string of ones starting from the LSB. This flag is used in the "Postcorrection Unit." To generate the F_2 flag vector for trailing-nine detection, UCR is examined for trailing Fs, or CR_1 is 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 designs compute the F_2 flag vector 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 "Postcorrection Unit," and therefore the trailing-nine

$CA_1 x 10^{EA_1}$	= 9	000050000000000	x 10^{85}	
− $CB_1 x 10^{EB_1}$	= 0	000200000000010	x 10^{100}	
$CA_S x 10^{EA_S}$	= 0	000200000000010	x 10^{100}	
− $CB_S x 10^{EB_S}$	= 9	000050000000000	x 10^{85}	
RSA	= 11			
LSA	= 4		L G R S	
CA'_2	= 2	000000000100000	0 5 0	
− CB'_2	= 0	000000000090000	5 0 0	
CA_3	= 2	000000000100000	0 5 0	
+ CB_3	= F	FFFFFFFFF6FFFF	A F F	
UCR	= 2	000000000006FFFF	B 4 F	
C_1	= 1	111111111100000	0 1 0	
F_1	= 0	000000000000000	0 1 F	
F_2	= 0	000000000011111		
CR_1	= 2	000000000009999	5 5 0	
+ INJ_COR	=		4 5 0	
$CR_2 x 10^{EB_S}$	= 2	000000000010000	x 10^{96}	

Fig. 9. Example of DFP subtraction with roundTiesToAway.

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

$$\begin{aligned} ADD: (flagADD_0)_i &= \big((UCR)_i \equiv 15\big) \vee \big((UCR)_i \equiv 9\big) \\ &\quad \wedge \big((C_1)_{i+1} \equiv 1\big), \\ SUB: (flagSUB_0)_i &= \begin{cases} \text{For i} = 4 \\ \big((UCR)_4 \equiv 15\big) \wedge (P_3 \equiv 0) \\ \vee\big((UCR)_4 \equiv 14\big) \wedge (P_3 \equiv 1) \\ \text{For i} = 5 \ldots 19 \\ \big((UCR)_i \equiv 15\big), \end{cases} \end{aligned} \tag{7}$$

where $(C_1)_{i+1}$ is the carry-out bit of digit position i, and P_3 is the block propagate of the G, R, and S positions shown in Fig. 7.

Rows 7-10 $(1 \le j \le 4)$

$$\begin{aligned} (flagADD_j)_i &= (flagADD_{j-1})_i \wedge (flagADD_{j-1})_{i-2^{j-1}}, \\ (flagSUB_j)_i &= (flagSUB_{j-1})_i \wedge (flagSUB_{j-1})_{i-2^{j-1}}, \\ F_2 &= \begin{cases} flagADD_4, & \text{if } EOP \text{ is ADD}, \\ flagSUB_4, & \text{if } EOP \text{ is SUB}. \end{cases} \end{aligned} \tag{8}$$

Synthesis results of this method compared to the K-S network in [9], which only has one set of flags for the "Postcorrection Unit," indicate only a 13.7 percent increase in area. Some techniques shown in [20] might help designers to further improve area or delay in the K-S network.

4.5 Postcorrection and Shift and Round

The temporary result generated from the K-S network requires postcorrection to convert the uncorrected result

UCR back to BCD to produce CR_1. The rules for performing this correction are defined below:

Rule 1: Enforced when performing effective addition:
Add "1010" (*correction of* -6) *to* $(UCR)_i$ *when* $(C_1)_{i+1}$ *is* 0
Rule 2: Enforced when performing effective subtraction:
If (*MSB of* $C_1 \equiv 1$) // *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" (correction of -6) to the above result in digit i if $(C_1)_{i+1} \oplus (F_1)_i^3 \equiv 0$

Else // the result is negative

1) Invert all sum bits
2) Add "1010" to the above result in digit i if $(C_1)_{i+1} \equiv 1$

Rule 1 is straightforward, since the precorrection 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 "Precorrection and Operand Placement 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 parallel 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 $(CA_3)_i < (CB_3)_i$. Therefore, a value of six should be subtracted from the sum digit, which is equivalent to adding a value of 10 to the digit position. Similarly, if the result is negative, all sum bits are inverted such that $CR_1 = CB_3 - CA_3$. Next, if $(C_1)_{i+1}$ is one, it means $(CB_3)_i < (CA_3)_i$. Therefore, a value of six is subtracted from, or equivalently 10 is added to, the sum digit at position i.

The "Shift and Round Unit" computes the final result significand based on the rounding mode and the sign of the result. If the MSD of CR_1 is zero, the "Shift and Round Unit" truncates the corrected result CR_1 from the "Postcorrection Unit" to obtain the final result significand. However, if the MSD of CR_1 is nonzero, an injection correction value is added to CR_1 to adjust the initial injection value, similar to the approach used by the injection-based method in binary arithmetic. This is because the injection value applied in the "Precorrection and Operand Placement Unit" is off by one digit if the MSD of CR_1 is nonzero. In this case, a second correction value, shown in Table 5, is added to CR_1. Adding the injection correction value from Table 5 to the injection value from Table 4 gives the overall injection value required when the MSD of CR_1 is nonzero.

As illustrated in Table 5, there are only two distinct nonzero injection correction values, and S is always zero for injection correction. Similar to Table 4, some injection correction values do not depend on $Sign_{inj}$ and this is denoted using "?". Since injection correction values are only needed if the MSD of CR_1 is nonzero, it is not possible to have another carry-out of the MSD due to adding injection correction values. To avoid the carry propagation network needed when adding the injection correction values, the F_2 flag vector, which is generated in the K-S network, is used to conditionally increment CR_1 via a row of parallel exclusive-OR gates.

TABLE 5
Injection Correction Values for Different Rounding Modes

$Sign_{inj}$	Rounding Modes	Injection Correction Value (G, R, S)
?	roundTowardZero	(0, 0, 0)
?	roundTiesToAway	(4, 5, 0)
?	roundTiesToZero[1]	(4, 5, 0)
?	roundTiesToEven	(4, 5, 0)
-	roundTowardPositive	(0, 0, 0)
+	roundTowardPositive	(9, 0, 0)
-	roundTowardNegative	(9, 0, 0)
+	roundTowardNegative	(0, 0, 0)
?	roundAwayZero[1]	(9, 0, 0)

[1] *Java BigDecimal library only*

4.6 Overflow, Sign, Backward Format Conversion, and Postprocessing

Overflow occurs when the addition or subtraction of two operands exceeds MAXFLOAT, the maximum representable DFP number in the destination format. Typically, the adder needs to check the carry-out of the MSD after rounding the corrected result to determine 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 result exponent ER_1 and the MSD of CR_1. The "Overflow Unit" also generates a signal to determine if the final result should be $\pm\infty$ or $\pm$MAXFLOAT based on the prevailing rounding mode and the sign of the result. Using this signal and the overflow flag, the final result is modified, if needed, in the "Postprocessing Unit."

The sign bit of the result SR_1 is determined by several factors. Equation (9) shows the normal case when no special cases or exceptions occur:

$$SR_1 = (\overline{EOP} \wedge SA_1) \vee \left(EOP \wedge \left(\overline{swap} \oplus SA_1 \oplus (C_1)_{16}\right)\right). \quad (9)$$

Since the sign bit is necessary in several other modules, such as the "Overflow Unit" and the "Shift and Round Unit," its value is determined as soon as possible. To quickly determine the sign of the result, all the equations for the special cases are duplicated with one set of equations assuming the MSD from the K-S network is zero and the other assuming it is one. After the addition, the carry-out of the MSD from the K-S network is used to quickly select the correct sign bit. This approach is similar to one used in the design of carry-select adders.

The "Backward Format Conversion Unit" encodes the sign bit, the exponent bits, and the significand digits to form the IEEE 754-2008 DPD-encoded result. Finally, the "Postprocessing Unit" handles special input operands in IEEE 754-2008, such as infinity, signaling and quiet NaNs, and results that trigger exceptions, such as overflow. Both our DFP adder and MFU do not need logic for the underflow exception because the DFP operations implemented in this paper do not generate results that are both subnormal and inexact.

4.7 Summary and Design Comparisons

In summary, the DFP operations presented in this paper are performed using the following steps:

- The "Forward Format Conversion Unit" extracts the sign bits, the biased exponents, and the significands from both operands, performs DPD to BCD conversion on both significands, and detects special values, such as NaN and infinity.
- The "OACSU" and the "Decimal Barrel Shifter" compute the left and right shift amounts, shift both significands, and generate the guard, round, and the sticky digits.
- The "Precorrection and Operand Placement Unit" places both significands based on the effective operation, injects values based on the rounding mode and the sign bit, and adjusts the significands based on the effective operation.
- The "K-S Network" generates the carry and sum vectors, and two flag vectors. One of the flag vectors F_1 handles increments in the postcorrection stage and the other F_2 handles carry propagation from the injection correction in the rounding stage.
- The "Postcorrection Unit" adjusts the uncorrected result UCR from the K-S network based on the sign of the result, the F_1 flag vector, and the carry-out of each digit of the result.
- The "Shift and Round Unit" uses the F_2 flag vector, which indicates a string of consecutive trailing nines starting from the LSD, to conditionally increment the corrected result if its MSD is nonzero. This is followed by truncation to obtain the final result significand.
- The "Backward Format Conversion Unit" combines the sign bit, the biased exponent, and the significand to form the result in IEEE 754-2008 format.
- The "Postprocessing Unit" conditionally replaces the result by a special result, such as NaN, $\pm\infty$, or $\pm$MAXFLOAT, based on the input operands, the overflow flag, the sign of the result, and the operation.

There are some major differences between the proposed DFP adder and the design presented in [9], which is the first published IEEE 754-2008-compliant DFP adder. First, the proposed design in parallel computes $|EA_1 - EB_1|$, LA_1, and LB_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 flag vectors for the postcorrection used in [9], there are two extra sets of flags $flagADD$ and $flagSUB$ to more quickly increment the corrected result and generate the overflow flag. There are also a few other minor optimizations including the internal use of the BCD encoding, instead of the excess-3 encoding, which leads to simpler circuitry in the "Precorrection and Operand Placement Unit" and a 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 6.

5 Decimal Floating-Point Multifunction Unit

There are several operations defined in IEEE 754-2008 that can use hardware available in the DFP adder. This section describes how six other DFP operations are integrated into the adder's data path with only a small increase in area and delay.

SameQuantum and Quantize are the only two decimal-specific operations defined in IEEE 754-2008. The operation $\mathrm{SameQuantum}(A, B)$ 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 "EAC" adder in the "OACSU." The original EAC adder 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.

$\mathrm{Quantize}(A, B)$ generates a DFP number that has the same value as A and the same exponent as B, unless rounding or an exception occurs. For example, $\mathrm{Quantize}(12{,}345 \times 10^{-4}, 1 \times 10^{-2}) = 123 \times 10^{-2}$ when the rounding mode is roundTiesToEven. Due to the length of the significand in the destination format, Quantize sometimes raises the 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 significand in the destination format. In this case, the invalid operation flag is raised and the output is a quiet NaN. $\mathrm{Quantize}(A, B)$ 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 DFP addition with CB_1 set to zero. 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 when $CB_1 \equiv 0$. To provide the correct sign bit and rounding action for Quantize in this case, the EOP is forced to "ADD" even when A is negative.

An example, shown in Fig. 10, illustrates how Quantize is realized in our DFP MFU. In the example, $EA_1 < EB_1$, so the two operands are swapped. Normally, in DFP addition, if CA_s is zero, no shift is performed on either operand because one of the significands is zero. However, in Quantize, if CA_s is zero, $RSA = EA_S - EB_S$. After shifting CB_s by RSA, both operands have a leading zero attached to the left of their MSD and the injection values of (5, 0) for the roundTiesToAway rounding mode is added to the right of the G digit of CA_s. This new value CA'_2, then has six added to each digit to produce CA_3. In the K-S network, UCR, C_1, F_1, and F_2 are generated. However, F_2 is not needed because the MSD of the result is always zero. Consequently, the injection correction step is not needed in Quantize.

$CA_1 x 10^{EA_1}$	= 9	5500000000000000	x 10^{85}
+ $CB_1 x 10^{EB_1}$	= 0	0000000000000000	x 10^{100}
$CA_S x 10^{EA_S}$	= 0	0000000000000000	x 10^{100}
+ $CB_S x 10^{EB_S}$	= 9	5500000000000000	x 10^{85}
RSA	= 15		
LSA	= 0	L	G R S
CA'_2	= 0	0000000000000000	0 5 0
+ CB'_2	= 0	0000000000000000	9 5 5
CA_3	= 6	6666666666666666	6 B 6
+ CB_3	= 0	0000000000000000	9 5 1
UCR	= 6	6666666666666667	0 0 7
C_1	= 0	0000000000000000	1 1 0
F_1	= 0	0000000000000000	0 0 1
F_2	= 0	0000000000000001	
CR_1	= 0	0000000000000001	0 0 1
$CR_2 x 10^{EB_S}$	=	0000000000000010	x 10^{100}

Fig. 10. Example of DFP quantize with roundTiesToAway.

RoundToIntegral(A) rounds a DFP number to an integer based on the prevailing rounding mode. For example, $\mathrm{RoundToIntegral}(12{,}345 \times 10^{-3}) = 12$ when the rounding mode is roundTiesToEven. 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, the "Special Operation Unit" examines the exponent of A and selects A as the final result if $EA_1 \geq bias$.

$\mathrm{Compare}(A, B)$ 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. $\mathrm{minNum}(A, B)$ returns A if $A \leq B$ and returns B if $B < A$, while $\mathrm{maxNum}(A, B)$ returns A if $A \geq B$ 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. If the numbers are in the same cohort, the standard allows returning either one of the operands. In our implementation, we follow the decNumberMin and decNumberMax functions defined in the decNumber library [21].

To implement Compare, minNum, and maxNum, the DFP MFU reuses the original DFP adder with *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 "Special Operation Unit" to the design. For minNum and maxNum, the "Special Operation Unit" directly selects one of the input operands as the result in a purely combination circuit design. In a pipelined design, the input operands move through the pipeline using staging registers and the "Special Operation Unit" selects the correct input operand for the result from one of these staging registers. As most of the functions in this unit are performed in parallel with the original data path, the only increase in the overall critical path delay is from a 64-bit 2-to-1 multiplexer in the "Special Operation Unit."

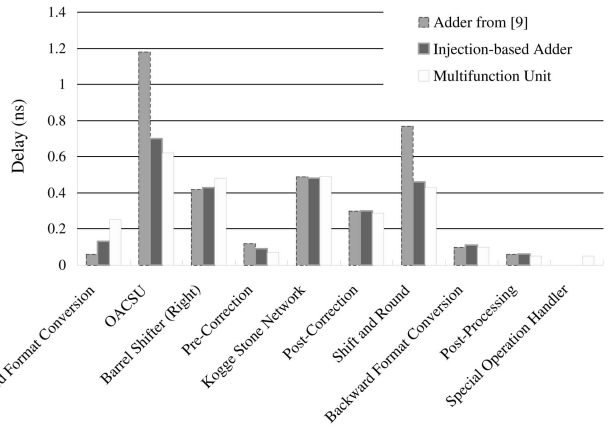


Fig. 11. DFP adder and MFU delay comparison.

6 Hardware Designs and Synthesis Results

Two DFP adders and the DFP MFU 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 was performed to ensure the correctness of the design. For a fair comparison, the adder design from [9] was extended to have the same functionality (i.e., handling both normal and special operands) as the proposed injection-based adder.

The DFP adders and MFU were synthesized using Synopsys Design Compiler and the 0.11 micron Gflx-p standard cell library from LSI Logic under normal operating conditions (1.2-V core voltage and $25\ ^{\circ}\mathrm{C}$ operating temperature). 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.

Figs. 11 and 12 compare the critical delay path and the area of the designs, respectively, when they are not pipelined. Table 6 compares the total area and delay of the three designs. As shown in Fig. 11, the proposed injection-based adder significantly reduces the delay in the "OACSU"

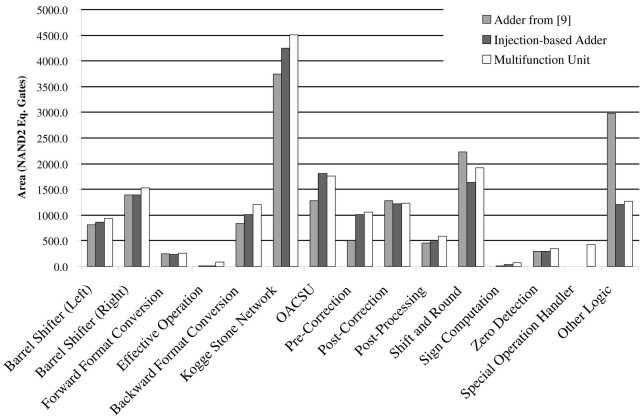


Fig. 12. DFP adder and MFU area comparison.

TABLE 6
DFP Adder and MFU Delay and Area Comparison

Designs	Delay (ns)	Delay (FO4)	Area (mm^2)	Area (NAND2 Eq. Gates)
Adder from [9]	3.50	63.6	0.1451	22443
Injection-based Adder	2.76	50.2	0.1428	22086
Multifunction Unit	2.84	51.6	0.1566	24233

and the "Shift and Round Unit," compared to the design presented in [9]. The proposed adder requires more area in the K-S network due to the generation of flag vectors for the "Postcorrection Unit" and the trailing-nine detection, and in the "Precorrection and Operand Placement Unit" due to the round-injection logic. However, the "Shift and Round Unit" is smaller and there is less random logic in the proposed adder than in the design from [9].

From Table 6, the proposed DFP adder has about 21 percent less delay and 1.6 percent less area than the design presented in [9]. The proposed MFU has 2.8 percent more delay and 9.7 percent more area than the proposed DFP adder. Compared to the theoretical FO4 inverter delay calculation for the double-precision BFP adder presented in [19], which uses a dual-path technique, the DFP injection-based adder has roughly 64 percent more delay.

To incorporate our DFP MFU into a processor's data path, it should be pipelined to achieve a cycle time that is less than or equal to the processor cycle time. To study potential implementations, our DFP MFU is pipelined using the *pipeline_design* command from Synopsys Design Compiler [22]. Results for pipeline depths from one to six stages are shown in Fig. 13. Although these synthesis results depend on the settings of the tool and its capabilities, they provide reasonable estimates of tradeoffs that can be made in area and delay for different pipeline depths. Fig. 13 indicates that a four-stage pipeline may be a good design option for the MFU. A six-stage pipeline can lead to a more aggressive critical path delay with some area overhead.

To demonstrate that the proposed pipeline strategy is feasible, pipelined four-stage and six-stage MFUs are implemented with pipeline stages manually included in the Verilog code. Synthesis results show that the four-stage MFU has a critical path delay of 0.91 ns (16.6 FO4 inverter delays) and area equal to 0.2386 mm^2 (36,911 NAND2 equivalent gates) and the six-stage MFU has a critical path delay of 0.71 ns (12.9 FO4 inverter delays) and area equal to 0.2953 mm^2 (45,681 NAND2 equivalent gates).

Table 7 shows a comparison of the latency of the DFP operations (except for sameQuantum) between our MFU, the fixed-precision version of the decNumber library (decDouble [21]), and the Intel's BID library (idflp64 [23]). The results in this table are taken from [21] and latencies for the sameQuantum operation are not included since they are not reported in [21]. A six-cycle pipelined DFP MFU, which can process a new operation every cycle, is used to compare against the software library. As can be seen from this table, our MFU is more than 20 times faster than either of the software libraries. As operations supported in the DFP MFU are quite common in commercial applications, the DFP MFU

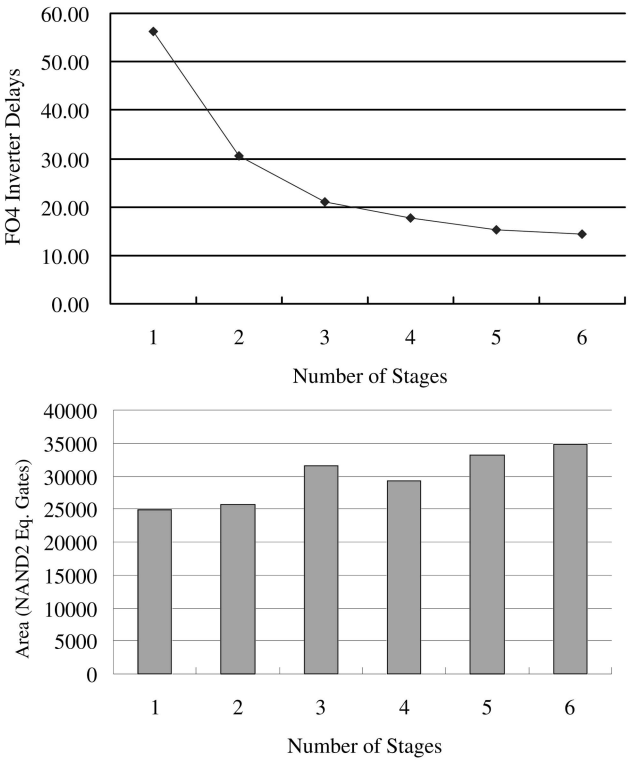


Fig. 13. Delay and area of DFP MFU for different pipeline depths.

can significantly improve the overall performance in target applications. For example, the benchmarks presented in [24] spend 10 percent to 40 percent of their execution time in operations supported by the DFP MFU.

7 Future Research

Analysis from [24] indicates that operands for DFP addition and subtraction often have the same exponent value in certain applications. This analysis also shows that DFP addition and subtraction often do not need rounding. To speed up DFP applications, it may be worthwhile to implement a variable-latency DFP adder or MFU with a fast path that avoids operand alignment when exponents are equal and avoids rounding when the final result is guaranteed to fit in the destination format. Although a variable-latency design may complicate the instruction scheduler, it may improve the overall performance of certain DFP applications.

A second potential research area is to explore internal DFP encodings that can further improve the performance of DFP operations. For example, rather than encoding and decoding DFP operands each operation, DFP operands can

TABLE 7
Performance of DFP Operations in Software and Hardware

	decDouble [21]	idflp64 [23]	DFP MFU
ADD	252	247	6
SUB	292	250	
COMPARE	133	146	
MaxNum/MinNum	155	153	
Quantize	121	182	
ToIntegralValue	135	131	

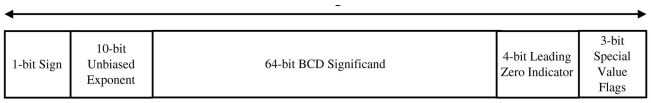


Fig. 14. Potential unpacked format for decimal64.

be stored in the register file in an "unpacked" format that includes the operand's sign, biased exponent, BCD-encoded significand, and bits that indicate if the number is a special value, such as NaN, infinity, or zero. To further improve performance, the number of leading zeros in the significand can also be stored in the register file. An example of this type of format is shown in Fig. 14 for "unpacked" decimal64 numbers. From the figure, only four bits are used to indicate the number of leading zeros in the significand and a total of only 18 extra bits are used to store the number in the unpacked format. Although the "unpacked" format increases the size of the register file and may make it necessary to perform conversion during load and store operations, it enables the leading zero detectors and the forward and backward conversion units to be removed from the MFU.

A third interesting research area is to investigate the potential costs and benefits of implementing other DFP operations, such as nextUp, nextDown, minNumMag, and maxNumMag, in the MFU.

As industry is interested in providing hardware support for decimal128, it is useful to study designs for decimal128 DFP MFUs and their area-delay tradeoffs. Although the techniques presented in this paper can be applied, a decimal128 MFU unit is more difficult to design than its decimal64 counterpart as wire can contribute significantly to the delay in current and future process technologies. Many subunits may be affected by this increasing delay.

8 Conclusion

In this paper, we have given an overview of DFP arithmetic in IEEE 754-2008 and discuss previous research on decimal fixed-point and floating-point addition. We also present novel hardware designs for a DFP adder and DFP MFU. We provide a detailed analysis of synthesis results and a comparison between a previous DFP adder design, our DFP adder design, and our DFP MFU design. Latency estimates from decimal software libraries are given to demonstrate the potential benefits of having hardware support for common DFP operations. We also discuss future optimizations that can be used to improve our designs.

Our DFP adder employs several novel techniques including parallel operand alignment, decimal injection-based rounding, and trailing-nine detection to reduce the critical path delay. The DFP adder is extended to a DFP MFU that support eight operations with only a minor increase in delay and area. Synthesis results show that the proposed adder design has 21 percent less delay and 1.6 percent less area than the DFP adder design in [9] and the DFP MFU only has about 2.8 percent more delay and 9.7 percent more area than the proposed DFP adder. Our DFP MFU is more than 20 times faster than decimal software libraries for common DFP operations.

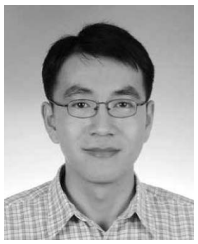
Acknowledgments

This work was done while the authors were with the University of Wisconsin-Madison. It was partially supported by IBM and the University of Wisconsin Graduate School.

References

[1] IEEE, *IEEE 754-2008 Standard for Floating-Point Arithmetic*, 2008.
[2] L. Eisen, J.W. Ward III, H.-W. Tast, N. Mading, J. Leenstra, S.M. Mueller, C. Jacobi, J. Preiss, E.M. Schwarz, and S.R. Carlough, "IBM POWER6 Accelerators: VMX and DFU," *IBM J. Research and Development*, vol. 51, no. 6, pp. 663-684, 2007.
[3] A.Y. Duale, M.H. Decker, H.-G. Zipperer, M. Aharoni, and T.J. Bohizic, "Decimal Floating-Point in z9: An Implementation and Testing Perspective," *IBM J. Research and Development*, vol. 51, nos. 1/2, pp. 217-228, 2007.
[4] C.F. Webb, "IBM z10: The Next-Generation Mainframe Microprocessor," *IEEE Micro*, vol. 28, no. 2, pp. 19-29, Mar./Apr. 2008.
[5] P.M. Kogge and H.S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. Computers*, vol. C-22, no. 8, pp. 786-793, Aug. 1973.
[6] G. Even and P.M. Seidel, "A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 638-650, July 2000.
[7] N. Burgess, "Prenormalization Rounding in IEEE Floating-Point Operations Using a Flagged Prefix Adder," *IEEE Trans. VLSI Systems*, vol. 13, no. 2, pp. 266-277, Feb. 2005.
[8] Sun Microsystem, *BigDecimal Class, Java 2 Platform Standard Edition 5.0, API Specification*, http://java.sun.com/j2se/1.3/docs/api/, 2004.
[9] J. Thompson, M.J. Schulte, and N. Karra, "A 64-Bit Decimal Floating-Point Adder," *Proc. IEEE CS Ann. Symp. VLSI (ISVLSI '04)*, pp. 297-298, Feb. 2004.
[10] L.-K. Wang and M.J. Schulte, "Decimal Floating-Point Adder and Multifunction Unit with Injection-Based Rounding," *Proc. 18th IEEE Symp. Computer Arithmetic (ARITH '07)*, pp. 56-68, June 2007.
[11] IEEE Inc., *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*, 1985.
[12] M.F. Cowlishaw, *Decimal Arithmetic FAQ: Part 1—General Questions*, http://www2.hursley.ibm.com/decimal/decifaq1.htm, 2003.
[13] R.K. Richards, *Arithmetic Operations in Digital Computers*. Van Nostrand, 1955.
[14] U. Grupe, *Decimal Adder*, US Patent 3,935,438, Jan. 1976.
[15] M.J. Adiletta and V.C. Lamere, *BCD Adder Circuit*, US Patent 4,805,131, Feb. 1989.
[16] H. Fischer and W. Rohsaint, *Circuit Arrangement for Adding or Subtracting Operands in BCD-Code or Binary-Code*, US Patent 5,146,423, Sept. 1992.
[17] M.S. Schmookler and A.W. Weinberger, "High Speed Decimal Addition," *IEEE Trans. Computers*, vol. 20, pp. 862-867, Aug. 1971.
[18] L.-K. Wang, "Processor Support for Decimal Floating-Point Arithmetic," PhD dissertation, Dept. Electrical and Computer Eng., University of Wisconsin-Madison, 2007.
[19] P.M. Seidel and G. Even, "Delay-Optimized Implementation of IEEE Floating-Point Addition," *IEEE Trans. Computers*, vol. 53, no. 2, pp. 97-113, Feb. 2004.
[20] A. Beaumont-Smith and C.-C. Lim, "Parallel Prefix Adder Design," *Proc. 15th IEEE Symp. Computer Arithmetic (ARITH '01)*, pp. 218-225, 2001.
[21] IBM Corporation, *The decNumber Library*, http://www2.hursley.ibm.com/decimal/decnumber.pdf, version 3.56, Apr. 2008.
[22] Synopsys, *Galaxy Design Platform*, http://www.synopsys.com, 2008.
[23] M. Cornea, C. Anderson, J. Harrison, P.T.P. Tang, E. Schneider, and C. Tsen, "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format," *Proc. 18th IEEE Symp. Computer Arithmetic (ARITH '07)*, pp. 29-37, 2007.

[24] L.-K. Wang, C. Tsen, M.J. Schulte, and D. Jhalani, "Benchmarks and Performance Analysis for Decimal Floating-Point Applications," *Proc. 25th IEEE Int'l Conf. Computer Design (ICCD '07)*, pp. 164-170, Oct. 2007.



Liang-Kai Wang received the BS degree in electronic engineering from the National Chiao Tung University, Hsinchu, Taiwan, in 1991, where he focused on audio signal processing for musical instruments. Dr. Wang received his MS degree in electrical engineering in 2003 and the PhD degree from the University of Wisconsin-Madison. He is currently with Advanced Micro Devices (AMD) Long Star Design Center, Austin, Texas. His research interests include high-performance ultra-low-power processor design, domain-specific processors, and decimal floating-point arithmetic. In the past, he worked at Intel, where he helped to develop a new methodology for testing the Intel PXA800F cellular processor and other cellular/PDA processors. He is a member of the IEEE.

Michael J. Schulte received the BS degree in electrical engineering from the University of Wisconsin, Madison, and the MS and PhD degrees in electrical engineering from the University of Texas at Austin. He is currently an associate professor at the University of Wisconsin-Madison, where he leads the Madison Embedded Systems and Architectures Group. His research interests include high-performance embedded processors, computer architecture, domain-specific systems, and computer arithmetic. He is a senior member of the IEEE.

John D. Thompson received the BS and MS degrees in computer engineering from the University of Wisconsin, Madison, in 2002 and 2003, respectively. He is a hardware engineer at Cray Inc., Chippewa Falls, Wisconsin. His current research interests include design verification techniques as well as memory and network system architecture and performance modeling.

Nandini Jairam received the bachelor's degree in electronics and communications engineering from the University of Madras, India, in 2001 and the master's degree from the University of Wisconsin in 2003. During her master's studies, she worked with Prof. Schulte on developing the algorithm for Decimal Floating-Point addition. Since 2003, she has been a component design engineer, designing and testing next-generation chipsets and graphics processors, within the Mobility Group at Intel Corp., Folsom, California.