2009 19th IEEE International Symposium on Computer Arithmetic

A Decimal Floating-Point Adder with Decoded Operands and a Decimal Leading-Zero Anticipator

Liang-Kai Wang
Advanced Micro Devices
Lone Star Design Center
7171 Southwest Parkway, Austin, TX 78735
Liang-Kai.Wang@amd.com

Michael J. Schulte
University of Wisconsin-Madison
Dept. of ECE
1415 Engineering Drive, Madison, WI 53706
schulte@engr.wisc.edu

Abstract

The IEEE 754-2008 Standard for Floating-Point Arithmetic was officially approved this year. One of the most important revisions to IEEE 754-1985 is the introduction of decimal floating-point (DFP) formats and operations. Since IEEE 754-1985 was revised, major microprocessor vendors have been working on hardware designs and software libraries for decimal arithmetic. Because the new standard has been approved, many software vendors are planning to adapt the new decimal formats into their applications. Therefore, it is important to investigate efficient algorithms and hardware designs for common DFP arithmetic operations to improve the performance of these applications. This paper presents a novel DFP adder with decoded operands and a decimal leading-zero anticipator (LZA). The DFP adder is based on a previous DFP adder design with several new features, including a new internal format, an improved operand pre-correction stage, and a novel decimal LZA to obtain better timing for decimal addition and subtraction. Synthesis results show that the new DFP adder is roughly 14% faster than the previous design.

1. Introduction

Although binary floating-point arithmetic is usually sufficient for scientific and statistical applications, it is not adequate for many commercial and monetary applications. Often, these applications use software libraries to perform decimal floating-point (DFP) arithmetic operations. Although these software libraries eliminate the error from representing decimal numbers in binary and correctly emulate decimal rounding, their execution times are slow for numerically intensive commercial applications [2]. Furthermore, decimal numbers in different software libraries are often represented in different formats [11, 13, 19]. Due to the growing importance of DFP arithmetic, specification for it are now included in the IEEE 754-2008 Standard.

Support for DFP arithmetic includes software libraries, such as the Java BigDecimal library [20], IBM's decNumber library [8], and Intel's Decimal Floating-Point Math library [10]. Much of the hardware support in the past focused on designs for decimal fixed-point arithmetic, such as [5, 14, 17]. Recently, many papers, including [21–23], discuss techniques to design IEEE 754-2008-compliant adders, and IBM has also started offering microprocessors supporting DFP arithmetic, as specified in the IEEE 754-2008 Standard [3, 4, 18, 25].

This paper presents a novel DFP adder with decoded operands and a novel decimal leading-zero anticipator (LZA). The adder presented in this paper is based on the algorithm presented by Wang *et al.* in [23, 24], with the goals of improving the overall timing and shortening the latency of decimal64 DFP addition/subtraction. Although some of the modules are derived from [23, 24], significant changes are made in several modules to improve the latency. Additionally, a novel decimal LZA is developed to ensure the DFP adder generates an accurate leading-zero count (LZC) of the result, which is essential for later decimal operations on decoded operands.

Unlike decimal addition, there have not been any publications related to the design of decimal LZAs. Most previous papers on LZAs focus on the design and optimization of binary LZAs [1, 6, 16].

The format of input and output operands used in this design is an internal format, which includes the LZC. The LZC is part of the internal format, since leading-zero detection is often on the critical path of common DFP operations if the LZC is not provided. For example, LZCs of both operands are required in DFP addition to align the operands before significand precorrection and addition [23, 24]. Instead of computing the LZC of each operand at the beginning of the operation, the overall timing improves if the

1063-6889/09 $25.00 © 2009 IEEE
DOI 10.1109/ARITH.2009.9

LZC of each input operand is provided, and each DFP operations generates the LZC, sign, exponent, and decoded significand of the result. In theory, the memory format for DFP operands can be either one of the DFP encodings specified in IEEE 754-2008 (*i.e.*, the decimal encoding or the binary encoding) since the choice of the memory format does not affect the adder design. However, having the memory format be the decimal coding of DPD numbers greatly simplifies the conversion process. The LZC can also be generated by the other DFP operations. The mechanism of generating the LZC of operands during memory load and other DFP operations is beyond the scope of this paper; however, the LZC of a decimal number can be generated using techniques similar to those presented in [23, 24].

In the rest of the paper, Section 2 presents the decoded format and the DFP adder design, and Section 3 discusses the design of the decimal LZA, which operates in parallel with significand addition. Section 4 provides the testing methodology and synthesis results, and Section 5 concludes this paper.

This paper refers to SX_Y, CX_Y, and EX_Y as the sign, significand, and exponent of a DFP number, respectively. X is A, B, or R to denote operands or the result. The subscript "$_Y$" is a digit that denotes the output of different modules. A binary string Q_i refers to the i^{th} bit in Q and $Q_{L,i}$ means the i^{th} bit in the L^{th} level of Q, where L is the level number in a binary tree. $[m, n]$ denotes a single signed decimal digit from *m* to *n*. $(N)_i^j$ refers to the j^{th} bit in digit position, i, in a decimal number N, where the least significant bit (LSB) and the least significant digit (LSD) have index 0. For example, $(CA_2)_2^3$ is bit three of digit two in the decimal significand CA_2. For a binary or decimal string, U^k or $U \overset{k}{..} U$ means a string of k $U's$.

2. Design of the Decimal Floating-Point Adder

2.1. Decoded Input and Output Formats

Instead of using an IEEE 754-2008 format, which encodes the sign, significand, exponent, and miscellaneous information, including whether the number is infinity or Not-a-Number (NaN), into 64 bits for the decimal64 format, a decoded decimal number in this paper is represented with an 83-bit internal format. Figure 1 shows the layout of this internal format, along with the decimal64 format in IEEE 754-2008 [9]. The exponent field is uncompressed in the internal format. The significand remains un-normalized and is encoded in BCD such that 64 bits are needed to represent the significand. In addition to the data portion, the internal format also carries four Special Value Flags (is-Infinity, is-Signaling NaN, is-Quiet NaN, and is-Zero) and four bits in decimal64 to record the LZC of the significand. The use of this internal format has the potential to reduce the latency of DFP addition and subtraction, as illustrated in the remainder of the paper.

2.2. Block Diagram

The DFP adder is based on the one presented by Wang *et al.* [23,24] with several improvements to reduce the latency. Figure 2 shows the block diagram of the new decimal adder.

In Figure 2, both significands go into Pre-correction units to generate $(CA)_i + 6$, $(CB)_i + 6$, $\overline{(CA)_i}$, and $\overline{(CB)_i}$, where $\overline{(CA)_i}$ and $\overline{(CB)_i}$ are the bit-inverted versions of each BCD digit in CA and CB. Therefore, each digit in $\overline{CA}$ (*i.e.*, $\overline{(CA)_i}$) is equal to $15 - (CA)_i$. Simultaneously, the effective operation (EOP) is determined and used to select the corrected significands. When EOP is addition, $(CA\_u)_i$ (modified CA when it serves as the augend) and $(CB\_u)_i$ (modified CB when it serves as the augend) are $(CA)_i + 6$ and $(CB)_i + 6$, and $(CA\_d)_i$ (modified CA when it serves as the addend) and $(CB\_d)_i$ (modified CB when it serves as the addend) are CA and CB. When EOP is subtraction, CA_u and CB_u are CA and CB, but CA_d and CB_d are $\overline{CA}$ and $\overline{CB}$, respectively.

While the significands are undergoing the correction, the Shift Amount unit determines if significand swapping is necessary ($swap \equiv 1$), the left and right shift amounts (*LSA* and *RSA*, respectively), and the temporary exponent (*eR_temp*). Since the LZC of the input operands is provided, leading-zero detectors (LZDs) are not needed in the Shift Amount unit.

The *swap* signal from the Shift Amount unit chooses the correct CA_2 and CB_2. The right and left shifters align the two resulting operands based on the shift amounts, RSA and LSA. Unlike the operands in the previous DFP adder [23, 24], the operands to the shifters have been recoded by the Pre-correction unit, so the digits that are shifted into the operands are corrected values. These corrected shifted-in values are 4'b0110 and 4'b0000 for CA_2 in effective addition and subtraction, respectively, and 4'b0000 and 4'b1111 for CB_2 in effective addition and subtraction, respectively.

The aligned significands enter the LZA to generate the LZC of the result and enter the Kogge-Stone Network (K-S Network) to generate the carry, un-corrected sum, and two sets of flags. These signals from the K-S Network then enter the Sign and Exception unit to determine the sign, the exception flags, and special values. In addition, they are fed into the Post-correction unit to convert the un-corrected sum back to a BCD-encoded significand. The temporary exponent (eR_temp), EOP, and carry-out of the MSD ($carry_{MSD}$), enter the Exponent unit to generate the final exponent value. The output from the Post-correction unit enters the Rounding unit and is conditionally incremented based on the prevailing rounding mode, the carry vector, and the flag vectors. The rounding unit uses decimal

Figure 1. Decoded Internal Format and Its Corresponding IEEE decimal64 Format

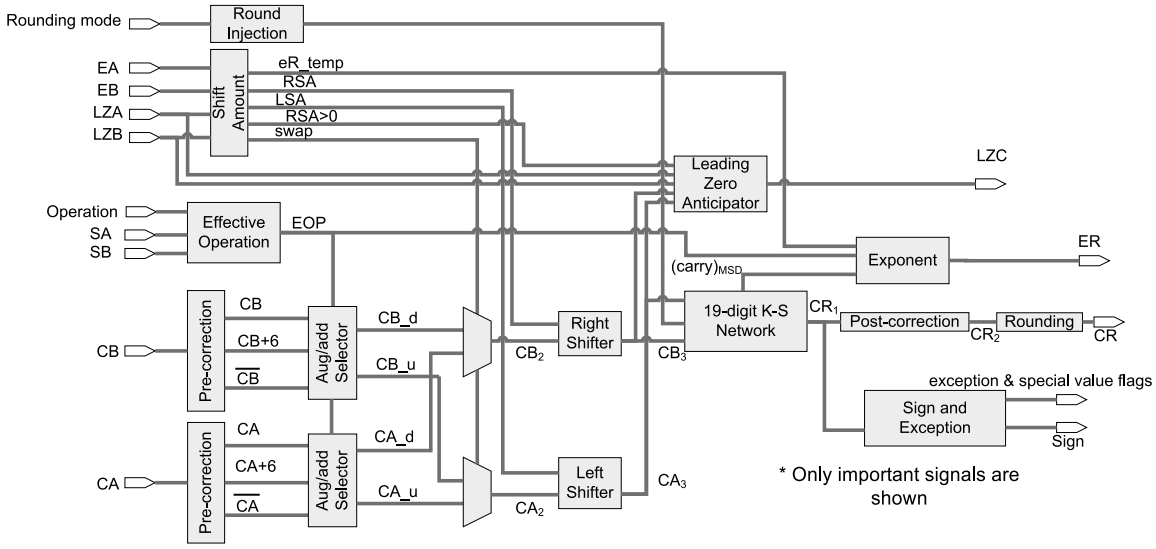

Figure 2. Block Diagram of the Proposed DFP Adder

injection-based rounding [23, 24].

Many of the building blocks in this DFP adder are from [23, 24]; however, they have been optimized to improve timing. For example, the Forward and Backward conversion units are no longer needed, since the input and output operands are encoded in the internal format. The Pre-correction unit is brought in front of the Swapping unit and duplicated for each operand, which removes the Pre-correction unit from the critical path. In addition, instead of using the LZD in the Shift Amount unit, which generates the LZC for both operands and is often on the critical path, the LZC is provided as an input signal, and an LZA is used to compute the LZC of the result, such that later decimal operations do not need to re-generate the LZC information. Because the LZA operates concurrently with the Kogge-Stone Network, the Post-correction unit and the Rounding unit, the adder's latency is improved.

Because many of the building blocks are similar to those proposed by Wang *et al.* in [23, 24], the following section focuses on the design of the decimal LZA.

3. Decimal Leading-Zero Anticipator

Unlike in binary arithmetic, LZA is needed in both effective addition and subtraction for decimal arithmetic to guarantee the LZC of the output is correct. However, it is worthwhile to note that LZA is only needed when the result after the decimal addition or subtraction is not rounded because the LZC is always zero when the result is rounded. To help readers understand the mechanism behind the LZA in this paper, Section 3.3 gives several examples. Sections 3.1 and 3.2 discuss the cases of effective addition and subtraction, respectively.

3.1. LZA for Effective Addition

Unlike in binary floating-point arithmetic, in IEEE 754-2008, DFP numbers are not normalized; therefore, the LZA must also predict the number of leading zero digits in effective addition. The number of leading zeros in the result is generally the LZC in CA_3 or CB_3, whichever is less. In other words, the Preliminary LZC, denoted as $PLZC_{add}$, is $min(LZC_{CA_3}, LZC_{CB_3})$. The only time correction is

needed is when there is a carry-out from the leading non-zero digit of the input operand that has a smaller LZC. In this case, the final LZC is obtained by decrementing $PLZC_{add}$ by one. For example, $0009998 + 0000002$ has a carry-out from the fourth most significant digit. The need for correction is indicated by the pattern $0^x pm^y gm?^z$, where $x > 0$ and $y \geq 0$; and pm and gm are the digit propagate and digit generate signals, respectively, and ? indicates a don't care condition for the digit.

This pattern can be observed using the following steps. First, note $(CA_3)_i$ and $(CB_3)_i$ are the operand digits after pre-correction, so $(CA_3)_i$ is equal to $(CA)_i + 6$ and $(CB_3)_i$ is still $(CB)_i$ when the effective operation is addition. Also, pm, gm, and zm bits are generated for each digit, where pm is the digit propagate and is equal to $((CA_3)_i + (CB_3)_i \equiv 15)$; gm is the digit generate signal and is equal to $((CA_3)_i + (CB_3)_i \geq 16)$; and, zm indicates if the result digit is zero and is equal to $((CA_3)_i + (CB_3)_i \equiv 6)$. These vectors are used to generate the final correction signal, Y_{add}, which is 1 when correction should be performed, based on the following equations:
Step 1:

$$\begin{aligned} z_{0,j} &= zm_j \wedge zm_{j-1} \\ p_{0,j} &= (zm_j \vee pm_j) \wedge pm_{j-1} \\ y_{0,j} &= gm_j \vee (zm_j \vee pm_j) \wedge gm_{j-1} \end{aligned}$$

Step 2 (through a four-level binary tree with L = 1 to 4):

$$\begin{aligned} z_{L,j} &= z_{L-1,2j+2^{L-1}} \wedge z_{L-1,2j} \\ p_{L,j} &= \left(z_{L-1,2j+2^{L-1}} \vee p_{L-1,2j+2^{L-1}}\right) \wedge p_{L-1,2j} \\ y_{L,j} &= y_{L-1,2j+2^{L-1}} \vee \left(z_{L-1,2j+2^{L-1}} \vee p_{L-1,2j+2^{L-1}}\right) \\ &\quad \wedge y_{L-1,2j} \\ Y_{add} &= y_{4,0} \end{aligned}$$

Note $\wedge$ and $\vee$ are logical-AND and logical-OR, respectively.

3.2. LZA for Effective Subtraction

Decimal subtraction of unnormalized positive operands *A* and *B* produces a result $Z = 0^j[-9, 9]^k$, where the most significant digit in the $[-9, 9]^k$ string is non-zero. Adapted from [1], to produce the correct LZC, the LZA needs an Encoding unit and a Correction unit. For this decimal adder, an parallel array of decimal digit adders is also needed. The Encoding unit includes a pre-encoding module, which converts BCD digits into strings of zeros and ones, and an encoding tree, which detects the position of the most significant non-zero digit in the string from the pre-encoding module. The leading zero count from the Encoding unit is denoted as the Preliminary LZC ($PLZC_{sub}$). The Correction unit consists of a pair of flag generation modules and correction trees to detect if a correction step is necessary on the $PLZC_{sub}$ value from the Encoding unit. Details for each unit and module are given below:

Table 1. Truth Table for Symbols in the LZA

$(W)_i$	$(carry)_i$	$(sum)_i$	Symbol Asserted
-9	0	0110	$s2_i, s9_i$
-8	0	0111	
-7	0	1000	
-6	0	1001	
-5	0	1010	$s2_i$
-4	0	1011	
-3	0	1100	
-2	0	1101	
-1	0	1110	$s1_i$
0	0	1111	$zero_i$
+1	1	0000	$g1_i$
+2	1	0001	
+3	1	0010	
+4	1	0011	
+5	1	0100	$g2_i$
+6	1	0101	
+7	1	0110	
+8	1	0111	
+9	1	1000	

Decimal Digit Adder: A decimal digit addition of the aligned and corrected significands, CA_3 and CB_3, is performed as

$$\begin{aligned} (W)_i &= (CA)_i - (CB)_i \\ &\Rightarrow (CA_3)_i + (CB_3)_i = ((carry)_i, (sum)_i) \end{aligned}$$

In effective subtraction, $(CA_3)_i = (CA)_i$ and $(CB_3)_i = (15 - (CB)_i)$, $(W)_i$ is the 5-bit digit result using manual subtraction, while $((carry)_i, (sum)_i)$ is the 5-bit result of $(CA_3)_i + (CB_3)_i$ in our design. $((carry)_i, (sum)_i)$ from each decimal digit adder is shared by the LZA for effective addition and subtraction to generate symbols for the correction signals, Y_{add} and Y_{sub}.

Encoding Unit: The Pre-encoding module in the Encoding unit converts W into strings of 0's and 1's. Each digit $(W)_i$ is examined to determine if it is -9, [-9, -2], -1, 0, 1, [2, 9], or 9, which are represented as $s9_i$, $s2_i$, $s1_i$, $zero_i$, $g1_i$, $g2_i$, or $g9_i$, respectively. These bit vectors are used to determine the leading zero count. Table 1 describes the relationship between these symbols, $(W)_i$, and $((carry)_i, (sum)_i)$ pairs.

Since this LZA is only needed in effective subtraction, in which $4'b0000 \leq (CA_3)_i \leq 4'b1001$ and $4'b0110 \leq (CB_3)_i \leq 4'b1111$, logic for the symbols in Table 1 can be further optimized in the hardware implementation. Also, although it is possible to generate these signals directly from $(CA_3)_i$ and $(CB_3)_i$, doing so usually creates too much out-

Table 2. Digit Patterns and the Corresponding Boolean Equations for Positive W

Digit Pattern of W	LZC Count	Substring Pattern	Boolean Equation
0^k[2,9][$-9,9$]^m	k	[2,9]	$g2_i$
0^k1[1,9][$-9,9$]^m	k	1[1,9]	$g1_i \wedge g1_{i-1} \wedge g2_{i-1}$
0^k10^t[0,9][$-9,9$]^m	k	10	$g1_i \wedge zero_{i-1}$
0^k10^t[-9,-1][$-9,9$]^m	$k+1$	10[1]	$g1_i \wedge zero_{i-1}$
0^k1[-8,-1][$-9,9$]^m	$k+1$	1[-8,-1][1]	$g1_i \wedge s1_{i-1} \wedge s2_{i-1} \wedge \overline{s9_{i-1}}$
0^k1(-9)^j[1,9][$-9,9$]^m	$k+j$	(-9)[1,9]	$s9_i \wedge g1_{i-1} \wedge g2_{i-1}$
0^k1(-9)^j[-8,-1][$-9,9$]^m	$k+j+1$	(-9)[-8,-1][1]	$s9_i \wedge s1_{i-1} \wedge s2_{i-1} \wedge \overline{s9_{i-1}}$
0^k1(-9)$^j0^t$[0,9][$-9,9$]^m	$k+j$	(-9)0	$s9_i \wedge zero_{i-1}$
0^k1(-9)$^j0^t$[-9,-1][$-9,9$]^m	$k+j+1$	(-9)0[1]	$s9_i \wedge zero_{i-1}$

Table 3. Digit Patterns and the Corresponding Boolean Equations for Negative W

Digit Pattern of W	LZC Count	Substring Pattern	Boolean Equation
0^k[-9,-2][$-9,9$]^m	k	[-9,-2]	$s2_i$
0^k(-1)[-9,-1][$-9,9$]^m	k	(-1)[-9,-1]	$s1_i \wedge s1_{i-1} \wedge s2_{i-1}$
0^k(-1)0^t[-9,-1][$-9,9$]^m	k	(-1)0	$s1_i \wedge zero_{i-1}$
0^k(-1)0^t[1,9][$-9,9$]^m	$k+1$	(-1)0[1]	$s1_i \wedge zero_{i-1}$
0^k(-1)[1,8][$-9,9$]^m	$k+1$	(-1)[1,8][1]	$s1_i \wedge g1_{i-1} \wedge g2_{i-1} \wedge \overline{g9_{i-1}}$
0^k(-1)9^j[-9,-1][$-9,9$]^m	$k+j$	9[-9,-1]	$g9_i \wedge s1_{i-1} \wedge s2_{i-1}$
0^k(-1)9^j[1,8][$-9,9$]^m	$k+j+1$	9[1,8][1]	$g9_i \wedge g1_{i-1} \wedge g2_{i-1} \wedge \overline{g9_{i-1}}$
0^k(-1)9$^j0^t$[-9,-1][$-9,9$]^m	$k+j$	90	$g9_i \wedge zero_{i-1}$
0^k(-1)9$^j0^t$[1,9][$-9,9$]^m	$k+j+1$	90[1]	$g9_i \wedge zero_{i-1}$

put load for $(CA_3)_i$ and $(CB_3)_i$ and affects the timing on the path through the K-S Network.

As in [1], the positive and negative strings of W are studied separately. For a positive W, Table 2, similar to Table 1 in [1], shows the digit pattern of W, number of leading zeros, the substring patterns, and the boolean equation of the substring patterns to detect the leading one in a decimal string. For example, a W string (000345678) matches the row $(0^k[2,9][-9,9]^m)$, and (001(-9)(-9)(-5)456) matches the row $(0^k$1(-9)^j[-8,-1][$-9,9$]$^m)$.

Some bit patterns that generate incorrect results (*i.e.*, off by one) are corrected later using the Correction unit. Additionally, the W string can be converted to a binary $P(W)$ string to determine the leading non-zero digit in W, where each bit, $P_i(W)$, in $P(W)$ is represented by

$$P_i(W) = g2_i \vee (g1_i \vee s9_i) \wedge \overline{s9_{i-1}}$$

Similarly, for a negative W (denoted as $\overline{W}$), a different set of patterns is used and shown in Table 3. The $\overline{W}$ string is converted to a binary $P(\overline{W})$ string, where each bit, $P_i(\overline{W})$, is

$$P_i(\overline{W}) = s2_i \vee (s1_i \vee g9_i) \wedge \overline{g9_{i-1}}$$

Like the binary LZA, it is possible to merge these two strings so that only one LZD is needed. In the case of $W > 0$, if the number of leading zeros is k or $k+1$, the digit prior to the most significant non-zero digit is always zero, whereas if the number of leading zeros is $k+j$ or $k+j+1$, the digit prior to the least significant (-9) is either 1 or -9 (*i.e.*, not zero.) It is similar in the $W < 0$ case. As a result, $P(W)$ and $P(\overline{W})$ can be combined into a new string P, where P_i is

$$P_i = \begin{array}{l} zero_{i+1} \wedge \left(\begin{array}{l} g2_i \vee s2_i \vee g1_i \wedge \overline{s9_{i-1}} \vee \\ s1_i \wedge \overline{g9_{i-1}} \end{array} \right) \vee \\ \overline{zero_{i+1}} \wedge \left(s9_i \wedge \overline{s9_{i-1}} \vee g9_i \wedge \overline{g9_{i-1}} \right) \end{array}$$

An LZD is used to determine the number of leading zeros in string P. This result is subject to correction and is a preliminary result, denoted as $PLZC_{sub}$ in this paper.

The location of the leading one in string P can be determined either by using a 17-bit priority encoder similar to the design by Hokenek and Montoye [6] or the technique developed by Oklobdzija [15]. Neither of these techniques requires a major design change because the BCD digits have been transformed into a binary string. As this path does not appear in the critical path in our adder design, a behavioral RTL description of the priority encoder is used to implement the Encoding tree.

Correction Unit: From Tables 2 and 3, a correction step is required for $W > 0$ when any of the following patterns occurs:

0^k10^t[-9,-1][$-9,9$]^m or 0^k1(-9)$^j0^t$[-9,-1][$-9,9$]^m or

[1]Need correction if merged with the other cases

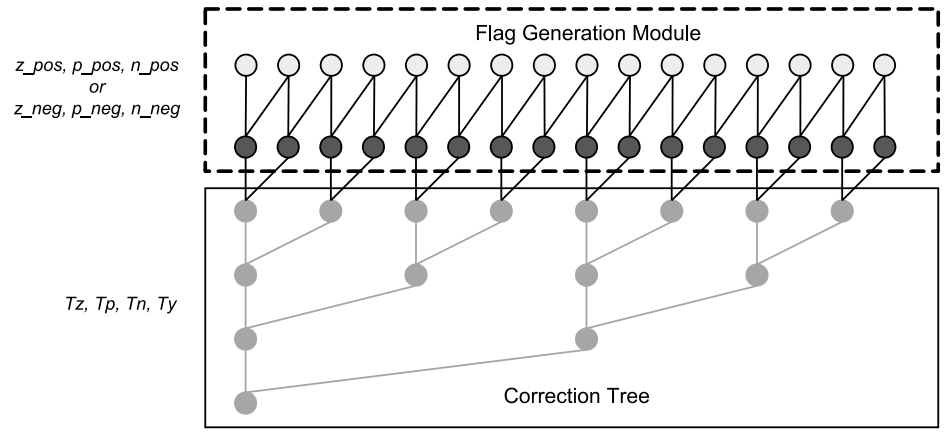

Figure 3. Tree Structure of the Correction Module for Effective Subtraction

0^k1[-8,-1]$[-9,9]^m$ or 0^k1(-9)^j[-8,-1]$[-9,9]^m$

These patterns can be consolidated into two general patterns, $0^{k+j}10^t[-9,-1][-9,9]^m$ and 0^{k+j}1[-8,-1]$[-9,9]^m$. To detect these patterns, three vectors, p_pos, n_pos, and z_pos, are generated, where p_pos_i indicates the occurrence of a leading one; n_pos_i indicates the occurrence of a terminate signal (*i.e.,* 0(-9) or [-8,-1]) and z_pos_i denotes a digit that is zero.

The equations for these vectors are shown below:

$$\begin{aligned} p\_pos_i &= g1_i \wedge \overline{s9_{i-1}} \vee s9_i \wedge \overline{s9_{i-1}} \\ n\_pos_i &= zero_{i+1} \wedge s9_i \vee s1_i \vee s2_i \wedge \overline{s9_i} \\ z\_pos_i &= (g1_i \vee s9_i) \wedge s9_{i-1} \vee zero_i \end{aligned}$$

Once these three vectors are computed, a Correction tree is used to determine whether correction is needed. Four variables, Tz, Tp, Tn, and Ty, are tracked in the Correction tree. The equations for each node of the tree are shown below:

$$\begin{aligned} Tz_{L,i} &= Tz_{L-1,2i+2^{L-1}} \wedge Tz_{L-1,2i} \\ Tp_{L,i} &= \begin{array}{l} Tz_{L-1,2i+2^{L-1}} \wedge Tp_{L-1,2i} \vee \\ Tp_{L-1,2i+2^{L-1}} \wedge Tz_{L-1,2i} \end{array} \\ Tn_{L,i} &= Tz_{L-1,2i+2^{L-1}} \wedge Tn_{L-1,2i} \vee Tn_{L-1,2i+2^{L-1}} \\ Ty_{L,i} &= \begin{array}{l} Ty_{L-1,2i+2^{L-1}} \vee Tz_{L-1,2i+2^{L-1}} \wedge \\ Ty_{L-1,2i} \vee Tp_{L-1,2i+2^{L-1}} \wedge Tn_{L-1,2i} \end{array} \end{aligned}$$

where $(Tz_{0,i}, Tp_{0,i}, Tn_{0,i}) = (z\_pos_i, p\_pos_i, n\_pos_i)$, $Ty_{0,i} = 0$ and $Y_{sub,pos} = Ty_{4,0}$.

Similarly, for negative W, a correction step is required if any of the following patterns is detected:

0^k(-1)0^t[1,9]$[-9,9]^m$ or 0^k(-1)9^j0^t[1,9]$[-9,9]^m$ or 0^k(-1)[1,8]$[-9,9]^m$ or 0^k(-1)9^j[1,8]$[-9,9]^m$

These four patterns can be consolidated into two patterns, 0^{k+j}(-1)$0^t[1,9][-9,9]^m$ and 0^{k+j}(-1)[1,8]$[-9,9]^m$. The pre-encoding equations are shown below:

$$\begin{aligned} p\_neg_i &= s1_i \wedge \overline{g9_{i-1}} \vee g9_i \wedge \overline{g9_{i-1}} \\ n\_neg_i &= zero_{i+1} \wedge g9_i \vee g1_i \vee g2_i \wedge \overline{g9_i} \\ z\_neg_i &= (s1_i \vee g9_i) \wedge g9_{i-1} \vee zero_i \end{aligned}$$

The tree structure for the negative W is the same as that for positive W, and $Y_{sub,neg} = Ty_{4,0}$. Figure 3 shows the tree structure for the correction module, including the flag generation module and the Correction tree. Since the sign, which is the carry-out from the most significant digit from the K-S Network, is generated more slowly than p's, n's, and z's, the positive and negative trees are not combined to avoid a critical path going through the Correction tree.

3.3. LZA Examples

To provide a better understanding of the mechanisms for LZA proposed in this paper, Figure 4 shows examples of LZA for eight digits with effective subtraction and addition, in which $PLZC_{add}$ and $PLZC_{sub}$ are the Preliminary LZC for effective addition and effective subtraction. Note CA_{shift} and CB_{shift} are never used in the hardware implementation; they are shown to provide a better understanding of the LZA and represent the significands after the operand alignment but without pre-correction. Figure 4(a) shows an example for effective subtraction in which there is a positive result and the correct signal, $Y_{sub,pos}$, is equal to one; Figure 4(b) provides an example for effective subtraction that produces a negative result and the correct signal, $Y_{sub,neg}$ is equal to zero; and, Figure 4(c) demonstrates an example for effective addition with the correction signal, Y_{add}, equal to one.

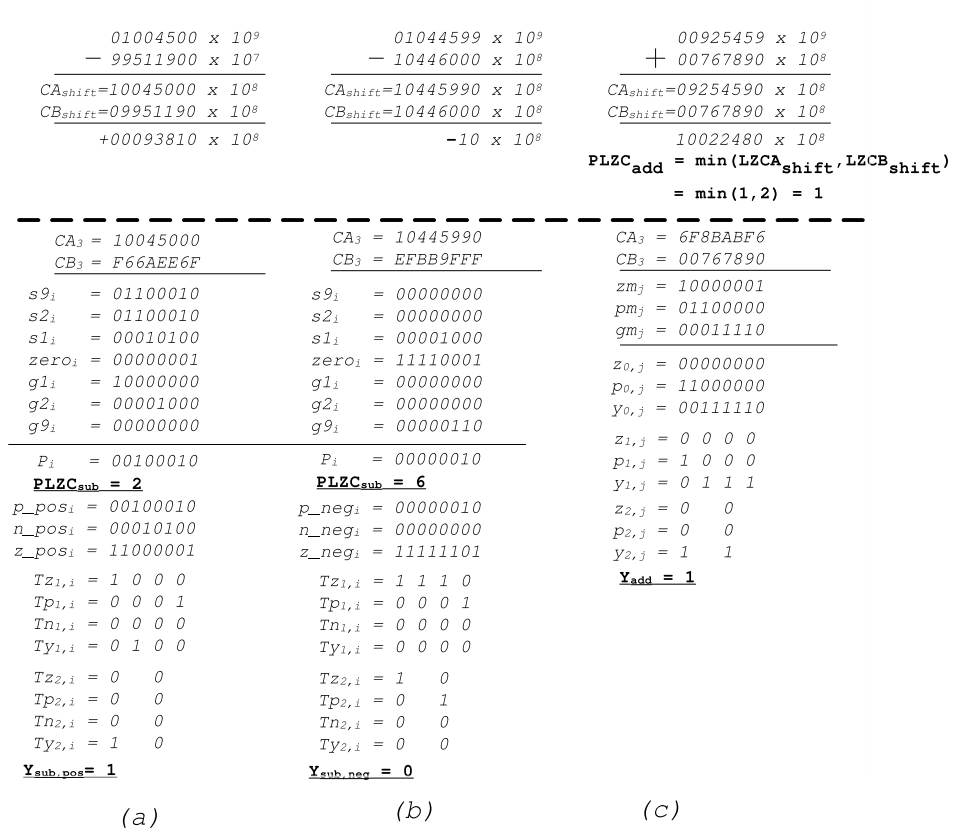

Figure 4. Examples of LZA with Effective Subtraction and Addition

3.4. Design of the LZA

Figure 5 shows the top-level block diagram of the LZA. The figure indicates an array of decimal digit adders to compute W. It also shows the LZC of the operand A, LZC_A, is corrected by subtracting the left shift amount, LSA, from the value. This corrected LZC, $LZCA_{shift}$, along with the $carry$ and sum from the decimal digit adders, and the LZC of the second operand, $LZCB$, enter the LZA for effective addition. In the other path, several miscellaneous signals enter a random logic block to generate a signal, *special_op*, to indicate the occurrence of zero, infinity, Signaling NaN, and Quiet NaN operands. Another signal, *normal_op*, is also generated from this random logic block to select the right LZA value at the end. The signals $carry$ and sum of W, $RSA > 0$, and *special_op* enter the LZA for effective subtraction. Note the LZA for subtraction is 17 digits wide to accommodate the case in which there is a one digit right shift and catastrophic cancellation occurs. The final LZC of the result is selected based on whether both operands are normal as well as the EOP.

Figure 6 shows the block diagram for the LZA for effective addition. In this unit, a flag generation module is used to produce gm, pm, and zm signals. Those signals then go into the initial merging module to generate vector signals, y, p, and z, which then enter the Correction tree for addition to generate the Y_{add} signal. The other side of the unit is composed of a 4-bit comparator to select the $PLZC_{add}$, a 4-bit decrementer to produce the result, lza_{minus}; and, a final multiplexer to select the correct LZC value.

Figure 7 presents the block diagram of the LZA for effective subtraction. As discussed in the previous section, this unit is composed of two modules, an Encoding unit, composed of a Pre-encoding module, an Encoding tree and additional logic, and a Correction Module, made of two flag generation blocks, Correction trees, and multiplexer. The encoding tree generates the Preliminary LZC ($PLZC_{sub}$) from the P signal. To handle catastrophic cancellation correctly, $PLZC_{sub}$ is decremented by one if the second operand, CB, is right-shifted. To reduce the latency, both $PLZC_{sub}$ and $PLZC_{sub} - (RSA > 0)$ are incremented, and the sign signal from the K-S network and the Y_{sub} signal from the Correction module are used to select the correct LZC for effective subtraction. In Figure 7, the R box is used to correctly handle infinity and NaNs. In this adder, the LZC is set to 15 if either input is NaN or infinity or the

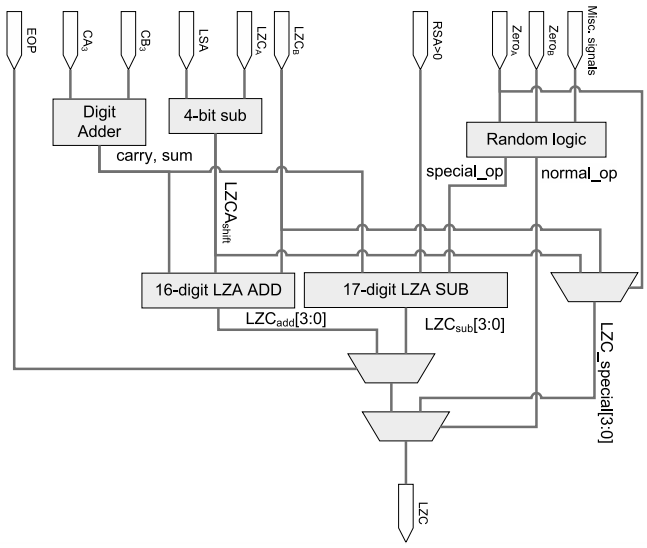

Figure 5. Overall LZA

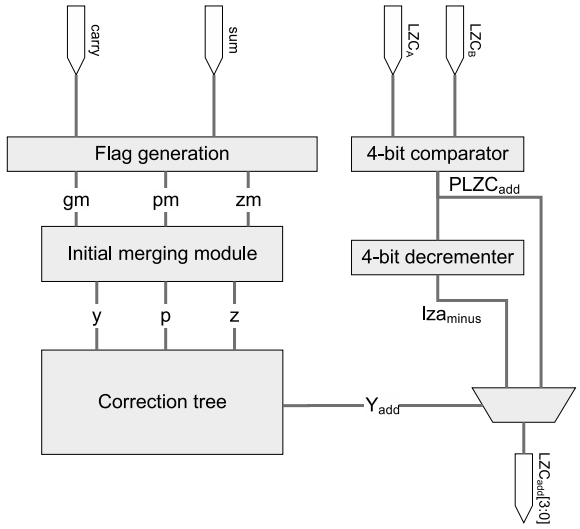

Figure 6. LZA for Effective Addition

result is zero.

4. Testing and Synthesis

4.1. Testing

The IBM decNumber library version 3.56 is used to verify the correctness of this DFP adder. To ensure the test cases cover as many scenarios as possible, the sign, exponent, and length and value of the significand of decimal operands are randomly generated before they are fed into the decNumberAdd and decNumberSubtract functions in the decNumber library. The random functions defined in the standard library in GNU *C* are used to generate those values, except for the exponent and the significand. The exponent is generated based on several special cases to test the correctness of LZA, The significand is generated using a modified version of the linear congruential random number generator [12]. This adder has undergone and successfully passed millions of random tests as well as the corner cases provided in IBM's test suite [7].

4.2. Synthesis Result

Both the DFP adder presented by Wang *et al.* in [23, 24] and the one in this paper are implemented in Verilog RTL, and only the inputs and outputs of the adders are registered. Both adders are synthesized using the TSMC 45nm bulk technology and an in-house CAD tool flow that integrates several industrial CAD tools, including the Synopsys Design Compiler (DC) for design compilation and IC Compiler (ICC) for cell placement. Only the results after ICC are reported because ICC provides more accurate timing and area information than design synthesis. Table 4 provides a comparison of synthesis results between the adder by Wang *et al.* [23, 24] and the adder in this paper. Since both designs use the same floorplan, the area utilization rate reflects how much of the area is used by each design. In other words, a higher area utilization rate means more area is used for cell placement.

Table 4. DFP Adder Delay and Area Comparison

Adder	Critial Path Delay (FO4)	Area Util. Rate	Cell Count
Adder in [23]	56	55%	10713
Proposed Adder	48	64%	12503
Change	-14%	+18%	+17%

Figure 8 shows the area profile of the adder after placement in both percentages and cell counts. As can be seen in the figure, the Kogge-Stone adder with flag tracing has the largest total area. This is followed by the LZA, the 19-digit Right Shifter, and the 16-digit Left Shifter.

The LZA is also synthesized by itself. The critical path delay, assuming all inputs of the LZA arrive simultaneously, is about 24 FO4 inverter delays. Figure 9 gives the area profile of the LZA in both percentages and cell count. It shows the LZA for Effective Subtraction is the largest contributor to the area of the total LZA. It occupies roughly 60% of the LZA.

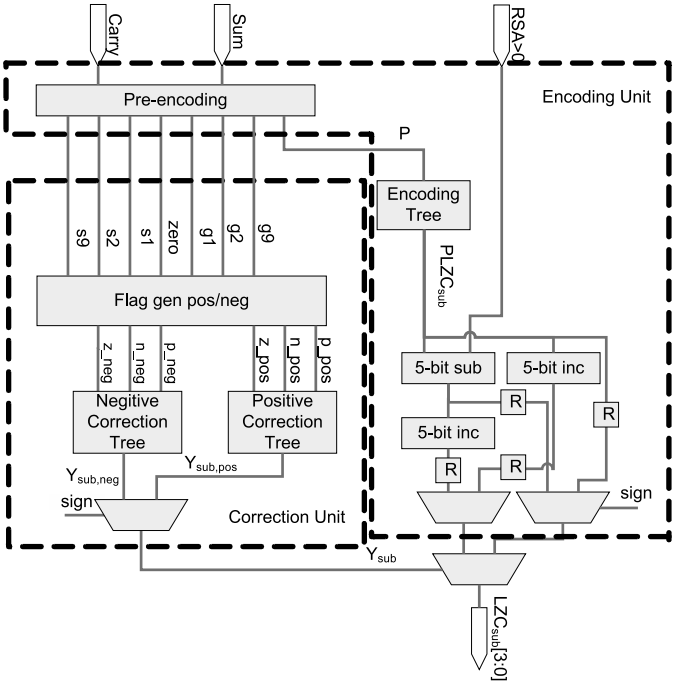

Figure 7. LZA for Effective Subtraction

5. Conclusion

In this paper, a new decimal floating-point adder, based on decoded operands and leading-zero anticipation, and a new decimal LZA are presented. The new DFP adder improves the timing of the previous design by 14% with an area increase of 18%. When synthesized by itself, the LZA has a maximum critical path delay of 24 FO4 inverter delays.

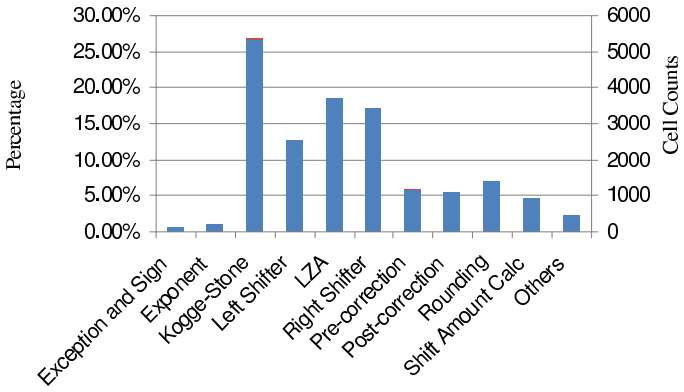

Figure 8. Area Profiling of the DFP Adder

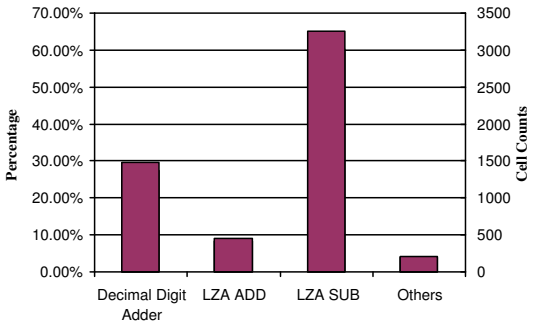

Figure 9. Area Profiling of the LZA Unit

6. Acknowledgement

The authors would like to thanks Steve Hejl, Derek Urbaniak, Carl Lemonds, and Eric Quinnel for reviewing the paper, and also Carl Lemonds and Derek Urbaniak for initiating and supporting the research. This paper was done when Michael Schulte was on sabbatical with AMD's Research and Advanced Development Labs.

References

[1] J. D. Bruguera and T. Lang. Leading-one prediction with concurrent position correction. *IEEE Transactions on Computers*, 48(10):1083–1097, Oct 1999.

[2] M. F. Cowlishaw. Decimal arithmetic FAQ: Part 1 - general questions. `http://www2.hursley.ibm.com/decimal/decifaq1.htm`, 2003.

[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 Journal of Research and Development*, 51(1/2):217–228, 2007.

[4] 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 Journal of Research and Development*, 51(6):663–684, 2007.

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

[6] E. Hokenek and R. K. Montoye. Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research and Development*, 34(1):71–77, 1990.

[7] IBM. General decimal arithmetic testcases. `http://speleotrove.com/decimal/dectest.html`.

[8] IBM Corporation. The decNumber library. `http://www2.hursley.ibm.com/decimal/decnumber.pdf`, April 2008.

[9] IEEE Inc. IEEE 754-2008 Standard for Floating-Point Arithmetic. New York, 2008.

[10] Intel Corp. Intel decimal floating-point math library. http://softwarecommunity.intel.com/articles/eng/3687.htm.

[11] JTC1/SC22/WG4. ISO/IEC 1989:2002- COBOL standard. http://www.cobolstandard.info/wg4/wg4.html, April 2002.

[12] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, section 1.2, pages 10–119. Addison-Wesley, Reading, Massachusetts, second edition, 10 Jan. 1973.

[13] Microsoft Corporation. C# language specification. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csspec%/html/CSharpSpecStart.asp, 2002.

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

[15] V. G. Oklobdzija. An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2:124 – 128, March 1994.

[16] M. S. Schmookler and D. G. M. Jr. Two state leading zero/one anticipator (LZA). *U.S. Patent #5,493,520*, Feb 1996.

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

[18] E. M. Schwarz, J. Kapernick, and M. Cowlishaw. Decimal floating-point support on the IBM z10 processor. *IBM Journal of Research and Development*, 53(1), 2009.

[19] Sun Microsystem. BigDecimal Class, Java Platform 1.1 API Specification. http://java.sun.com/products/archive/jdk/1.1/, 1999.

[20] Sun Microsystem. BigDecimal class, Java 2 platform standard edition 5.0, API specification. http://java.sun.com/j2se/1.3/docs/api/, 2004.

[21] 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,*, Feb 2004.

[22] C. Tsen, S. Gonzalez-Navarro, and M. J. Schulte. Hardware design of a binary integer decimal-based floating-point adder. *Proceedings of the* 25^{th} *IEEE International Conference on Computer Design*, 2007.

[23] L.-K. Wang and M. J. Schulte. Decimal floating-point adder and multifunction unit with injection-based rounding. *in Proceedings of the* 18^{th} *IEEE Symposium on Computer Arithmetic, Montpellier, France*, June 2007.

[24] L.-K. Wang, M. J. Schulte, J. D. Thompson, and N. Jairam. Hardware designs for decimal floating-point addition and related operations. *IEEE Transactions on Computers*, 58(3), March 2009.

[25] C. F. Webb. IBM z10: The next-generation mainframe microprocessor. *IEEE Micro*, 28(2):19–29, March/April 2008.