High-Speed Multioperand Decimal Adders

Robert D. Kenney and Michael J. Schulte, *Senior Member*, *IEEE*

Abstract—There is increasing interest in hardware support for decimal arithmetic as a result of recent growth in commercial, financial, and Internet-based applications. Consequently, new specifications for decimal floating-point arithmetic have been added to the draft revision of the IEEE-754 Standard for Floating-Point Arithmetic. This paper introduces and analyzes three techniques for performing fast decimal addition on multiple binary coded decimal (BCD) operands. Two of the techniques speculate BCD correction values and correct intermediate results while adding the input operands. The first speculates over one addition. The second speculates over two additions. The third technique uses a binary carry-save adder tree and produces a binary sum. Combinational logic is then used to correct the sum and determine the carry into the next more significant digit. Multioperand adder designs are constructed and synthesized for four to 16 input operands. Analyses are performed on the synthesis results and the merits of each technique are discussed. Finally, these techniques are compared to several previous techniques for high-speed decimal addition.

Index Terms—Computer arithmetic, decimal arithmetic, multioperand adders, hardware designs.

1 Introduction

Decimal computer arithmetic has been around since the beginning of modern computing. One of the earliest digital computers, the ENIAC, became operational in 1945 and used a decimal base for its arithmetic operations [1]. Calculators also provide direct support for decimal arithmetic. Though similar to the way humans perform arithmetic, decimal arithmetic in general purpose computers was quickly replaced by binary arithmetic, which is a more natural approach in digital circuits. With hardware being such a precious commodity in early computers, representing only 10 decimal numbers with four bits in a binary coded decimal (BCD) format was much less efficient than representing 16 binary numbers with the same four bits. Furthermore, decimal arithmetic operations are typically more complex and slower than binary arithmetic operations since they need to handle a wider range of digits, carries across both bit and digit boundaries, and the correction of invalid result digits.

Recently, support for decimal arithmetic has received increased attention due to the growing importance of financial, commercial, and Internet-based applications which cannot tolerate errors from converting between decimal and binary formats. Since many decimal numbers, such as 0.2, cannot be exactly represented in binary, these applications often store data in a decimal format and process data using decimal arithmetic software [2]. Although decimal arithmetic software eliminates conversion errors, it is typically 100 to 1,000 times slower than binary arithmetic implemented in hardware [2].

Due to the growing importance of decimal arithmetic, specifications for it have recently been added to the draft revision of the IEEE 754 Standard for Floating-Point Arithmetic [3]. The current IEEE 754 Standard, which was adopted in 1985, specifies support for binary floating-point arithmetic and is implemented on most general-purpose computers [4]. It is anticipated that, once the revised IEEE 754 Standard has been officially ratified, computer companies will begin to incorporate hardware support for decimal floating-point arithmetic on their processors for financial, commercial, and Internet-based applications. As transistor costs continue to decrease, the opportunity exists to incorporate high-speed decimal arithmetic units on future processors.

This paper introduces and analyzes various techniques for high-speed multioperand decimal addition. Multioperand addition is important because it often forms the core of other arithmetic operations, such as multiplication and division [5]. Consequently, efficient multioperand decimal addition is essential to the implementation of fast decimal multipliers and dividers. Multioperand decimal addition may also be useful for quickly summing large amounts of decimal data. Section 2 gives an overview of two-operand decimal addition, which has been the focus of most previous research in this area. Section 3 introduces three techniques for multioperand addition and discusses their hardware costs and worst-case delay paths. Section 4 presents area and delay estimates for the multioperand decimal adders, along with estimates for multioperand binary adders. Section 5 describes related work on decimal addition. Section 6 gives our conclusions. This paper is an extension of our research presented in [6]. Additional information on decimal arithmetic is available from http://mesa.ece.wisc.edu and http://www2.hursley.ibm.com/decimal/.

In the remainder of this paper, decimal numbers are assumed to be in Binary Coded Decimal (BCD) format. Subscripts next to constants are used to denote the base of the constant. For example, 1001_2 represents the binary number equal to nine. When no subscript is given next to a constant, a decimal base is implied. An upper case variable (e.g., A_i) denotes an entire operand word. A lower case

- *R.D. Kenney is with IBM Corp. and can be reached at 4023 Valley Ridge Rd., Middleton, WI 53562. E-mail: rdkenney@uwalumni.com.*
- *M.J. Schulte is with the Department of Electrical and Computer Engineering, University of Wisconsin-Madison, 1415 Engineering Dr., Madison, WI 53706. E-mail: schulte@engr.wisc.edu.*

Manuscript received 3 Sept. 2004; revised 4 Feb. 2005; accepted 5 Apr. 2005; published online 15 June 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0285-0904.

A	0010 0101 0111 0101
B	+ 0011 0100 0010 1000
Sum 1	0101 1001 1001 1101
Correction 1	+ 0000 0000 0000 0110
Sum 2	0101 1001 1010 0011
Correction 2	+ 0000 0000 0110 0000
Sum 3	0101 1010 0000 0011
Correction 3	+ 0000 0110 0000 0000
Result	0110 0000 0000 0011

Fig. 1. Example BCD addition of $2,575 + 3,428 = 6,003$ with iterative correction.

A	0010 0101 0111 0101
B	0011 0100 0010 1000
Speculated Correction	+ 0110 0110 0110 0110
Sum	0111 0111 0011 1011
Carry	+ 0010 0100 0110 0100
Compressed Sum and Carry	1100 0000 0000 0011
Final Correction	- 0110 0000 0000 0000
Result	0110 0000 0000 0011

Fig. 2. Example BCD addition of $2,575 + 3,428 = 6,003$ with speculative correction.

variable (e.g., a_i) denotes a single digit in that operand. A digit referenced with brackets (e.g., $a_i[4]$) denotes a single bit in that digit.

2 Two-Operand BCD Addition

In BCD format, each decimal digit is represented using four bits, where the bit patterns 0000_2 to 1001_2 represent decimal digits 0 to 9 and the bit patterns 1010_2 to 1111_2 are invalid. In the most straightforward approach to BCD addition, two BCD digits are added together and the resulting sum is examined. If the sum is greater than nine, a correction value of six is added to the sum. Adding six skips the invalid bit patterns and yields the correct BCD sum digit and a carry-out of the digit [7]. For example, to compute $7 + 7 = 14$ in BCD, the two sevens are added to produce 1110_2. Since this sum is greater than 9, a correction of $6 = 0110_2$ is added to 1110_2, which yields a carry-out of 1 and a sum of 0100_2, which corresponds to the correct BCD representation of $14 = 0001\ 0100_2$.

Fig. 1 illustrates the difficulty in performing a word-wide BCD addition, using the example $2,575 + 3,428 = 6,003$. This addition cannot be completed with one word-wide binary carry-propagate addition and correction. After the first carry-propagate addition of the two input operands, all resulting sum digits in Sum 1 potentially need to be corrected. In this example, only the 1101_2 in Sum 1 needs to be corrected since it is greater than nine. The addition of 0110_2 to 1101_2 in Correction 1 causes a carry to be passed to the next more significant digit. Now, the 1010_2 in Sum 2 needs to be corrected using Correction 2. This creates 1010_2 in Sum 3, which needs to be corrected. Finally, the correct BCD sum is produced in Result. In this example, four decimal carry-propagate additions are performed. To avoid repetitive word-wide additions and corrections, the least significant digits of A and B can be added, compared to nine, and corrected before proceeding to the addition of the next pair of digits with the correct carry-in. This approach, however, causes digit additions to be performed sequentially.

One way to improve the performance of BCD addition is to speculate that the sum for each pair of digits will be greater than nine [7]. This is done by adding six along with each pair of digits from the original input operands. As a result, carries between 4-bit digits now correspond to the value 10, rather than 16. The advantage of this approach is that the correct carries between digits are generated during the first addition. The resulting sum digits need to be corrected by subtracting six modulo 16 (i.e., adding 1010_2) if the carry-out is zero. This approach is illustrated in Fig. 2. First, the Speculated Correction, 6666, is added with A and B using binary carry-save addition to produce Sum and Carry. A word-wide binary carry-propagate addition is then performed to obtain Compressed Sum and Carry. After this, the Final Correction is subtracted using only digit-wide subtractions since all sum digits are guaranteed to be at least six.

One variant of decimal addition, called *direct decimal addition*, is proposed by Schmookler and Weinberger in [8]. This technique uses combinational logic to produce digit generate and propagate signals for each pair of BCD digits. The digit propagate and generate signals are then sent to carry-lookahead logic, which is used to compute digit carries in parallel. Finally, the digit carries and additional carry-lookahead logic within each digit are used to quickly produce the sum digits. The advantage of this method is that the combinational logic directly computes the correct sum digits, without the need for corrections.

Multioperand decimal addition can be performed by repetitively using two-operand decimal addition to add each operand. This approach, however, is very slow since each addition has carry propagation. A faster approach, which is introduced in this paper, is to use binary carry-save addition to compute intermediate results. When several operands are added together, the intermediate results are kept in binary carry-save format, delaying the carry-propagate addition until the end. The multioperand decimal addition algorithms introduced in this paper speed up the process of decimal addition when multiple BCD operands are added together. These algorithms are fundamentally different from multioperand binary addition algorithms [9], [10] since the sum and carry digits need to be corrected to ensure that proper BCD results are produced.

3 Proposed Techniques for Multioperand Decimal Addition

In this section, we introduce and analyze three techniques for performing fast decimal addition on multiple BCD operands. Two of the techniques speculate BCD correction values and correct intermediate results while adding the input operands. The first technique, Single Correction Speculation, speculates over one addition. The second technique, Double Correction Speculation, speculates over two additions. The third technique, Nonspeculative Addition, uses a binary carry-save adder tree [11] and produces a binary sum. Combinational logic is then used to correct the sum and determine the carry into the next more significant digit. The designs are first described for 1-digit multioperand decimal adders. We then explain how multiple 1-digit multioperand adders can be used to form word-wide multioperand decimal

```
1) s1 + c1 = a0 + a1;              // using simplified carry-save addition
2) for (i = 2; i < m; i++)         // for each of the remaining operands
       if (ci-1[4]==1)   si + ci = si-1 + ci-1 + (ai + 6);
       else              si + ci = si-1 + ci-1 + ai;
3) sm + cm = sm-1 + cm-1 + sc;     // speculation correction (sc) from Table 1
4) z' = sm + cm;                   // 4-bit carry-propagate addition compressing sm and cm
5) {co,z} = z' + f;                // f is the final correction from Table 2
```

Fig. 3. Single Correction Speculation Algorithm.

adders. In the following discussion, m denotes the number of input operands and n denotes the number of digits in each input operand.

3.1 Single Correction Speculation

The algorithm for Single Correction Speculation is shown in Fig. 3. The algorithm is called Single Correction Speculation since we implicitly speculate that the addition $a_0 + a_1$ does not need to be corrected. If a carry out of the current digit position occurs, this speculation is incorrect and a correction value of six is added along with the next input operand. With Single Correction Speculation, BCD digits from the first two input operands, a_0 and a_1, are added using simplified binary carry-save addition to produce a 4-bit sum digit, s_1, and a 4-bit carry digit, c_1, such that $s_1 + c_1 = a_0 + a_1$. The 4-bit simplified binary carry-save addition is performed using four half adders that operate in parallel to add corresponding bits from a_0 and a_1 and produce sum and carry vectors, s_1 and c_1, respectively. Using a simplified binary carry-save adder, which adds two input digits, instead of a conventional carry-save adder, which adds three input digits, allows there to be one correction for each addition. This simplifies the final correction logic and results in a less complex overall design.

When performing carry-save addition, each carry digit is shifted one bit position to the left relative to the corresponding sum digit. Thus, each sum digit, s_i, has bit positions $s_i[3:0]$, while each carry digit, c_i, has bit positions $c_i[4:1]$. Although each input digit, a_i, is a BCD digit with $0 \leq a_i \leq 9$, the sum and carry digits, s_i and c_i, are 4-bit values, where, in general, $0 \leq s_i \leq 15$ and $0 \leq c_i \leq 30$. The range of c_i is twice as large as that of s_i since it is shifted one bit position to the left.

If the most significant bit of the first carry digit, $c_1[4]$, is equal to one, then a carry out of the current digit position has occurred. In this case, a correction value of six is added to compensate for the fact that $c_1[4]$ has a weight of 16, while a decimal carry has a weight of 10. This can be expressed mathematically as

$$\begin{aligned} c_1[4:1] &= 16c_1[4] + 8c_1[3] + 4c_1[2] + 2c_1[1] \\ &= 6c_1[4] + (10c_1[4] + 8c_1[3] + 4c_1[2] + 2c_1[1]). \end{aligned} \quad (1)$$

Thus, $c_1[4]$ can be treated as a decimal carry, with a weight of 10, if a correction value of 6 is added when $c_1[4] = 1$.

To keep the addition of the correction value off the critical delay path, the correction value is added to the BCD digit of the next input operand, a_2, in advance and $c_1[4]$ is used to select between a_2 and $a_2 + 6$ as the next value to be added. A similar process continues for m-2 iterations ($2 \leq i < m$), until all m input operands are added with appropriate correction values. Each iteration, the most significant bit of the carry digit in the previous iteration, $c_{i-1}[4]$, is examined. If $c_{i-1}[4]$ is one, then a carry out of the current digit has occurred and $a_i + 6$ is added to s_{i-1} and c_{i-1} using carry-save addition to produce $s_i + c_i = s_{i-1} + c_{i-1} + (a_i + 6)$. Otherwise, no correction is needed and a_i is added to s_{i-1} and c_{i-1} to produce $s_i + c_i = s_{i-1} + c_{i-1} + a_i$.

At the end of the algorithm,

1. a speculation correction value, sc, is added to s_{m-1} and c_{m-1} based on $c_{m-1}[4]$,
2. a 1-digit carry-propagate addition is performed to compress the sum and carry digits to obtain a 5-bit preliminary sum, $z'[4:0] = s_m + c_m$,
3. the preliminary sum is corrected to produce z, and
4. a digit carry, co, is produced.

In Fig. 3, {co, z} corresponds to the 5-bit value obtained by concatenating co and z. It is useful to note that the corrections for the first m-1 iterations compensate for carries out of the current digit position, while the final corrections also help ensure that the final result is a valid BCD digit.

The algorithm produces the correct result since each time a carry-out of the current digit occurs (i.e., $c_i[4] = 1$), a correction value of six is added, which allows the digit carries to maintain a value of 10, rather than 16. This is expressed mathematically as

$$\begin{aligned} c_i[4:1] &= 16c_i[4] + 8c_i[3] + 4c_i[2] + 2c_i[1] \\ &= 6c_i[4] + (10c_i[4] + 8c_i[3] + 4c_i[2] + 2c_i[1]). \end{aligned} \quad (2)$$

Thus, each time $c_i[4] = 1$, six is added to the current digit position to correct the sum.

Since corrections are performed after each carry-save addition, the final correction steps are fairly simple, as shown in Table 1 and Table 2. Table 1 determines the correction value needed due to adding the last input operand. If $c_{m-1}[4] = 1$, then $sc = 6$ to correct for the carry out of the current digit position; otherwise, $sc = 0$. Table 2 determines the final correction value needed for Step 5 of Fig. 3. The logic that implements Table 2 examines the last two carry bits, $c_m[4]$ and $z'[4]$, and the preliminary sum digit, $z'[3:0]$. A value of six needs to be added to the current digit position when $c_m[4] = 1$, when $z'[4] = 1$, or when $z'[3:0] \geq 10$. Thus, the value of f, which corrects for all of these cases is:

$$f = 6 \times (c_m[4] + z'[4] + zge10), \quad (3)$$

where $zge10 = 1$ when $z'[3:0] \geq 10$. The final row of Table 2 cannot occur since $c_m[4]$ and $z'[4]$ each has a weight of 16 and $s_{m-1} + c_{m-1} + sc \leq 36$.

Fig. 4a shows an example of performing the 1-digit multioperand decimal addition $9 + 8 + 7 + 6 + 5 = 35$ using the Single Correction Speculation Algorithm, where carries

TABLE 1
Speculation Correction Selection for Step 3 of Fig. 3

$c_{m-1}[4]$	sc
0	0000_2 (+0)
1	0110_2 (+6)

TABLE 2
Correction Selection for Step 5 of Fig. 3

$c_m[4]$	z'[4]	z'[3:0]	f
0	0	z'[3:0] < 10	0
0	0	z'[3:0] ≥ 10	6
0	1	z'[3:0] < 10	6
0	1	z'[3:0] ≥ 10	12
1	0	z'[3:0] < 10	6
1	0	z'[3:0] ≥ 10	12
1	1	z'[3:0] < 10	12
1	1	z'[3:0] ≥ 10	N/A

out of the current digit position are shown in bold. When the algorithm completes, $z = 5$. Since there are a total of three carries out of the current digit position with a value of 1 (i.e., $c_1[4] = 1$, $c_4[4] = 1$, and $co = 1$), the correct result of 35 is produced.

Fig. 5 shows the block diagram of a 1-digit, 5-operand Single Correction Speculation Adder, where CSA and CPA correspond to Carry-Save Adder and Carry-Propagate Adder, respectively. Each multiplexer selects a_i when $c_{i-1}[4]$ is zero and $a_i + 6$ when $c_{i-1}[4]$ is one. Since a_i is a 4-bit BCD digit in the range of 0 to 9, $a_i + 6$ is in the range of 6 to 15. Thus, $a_i + 6$ is represented using just 4 bits and is computed from a_i using simple two-level logic [13]. Since the values for $a_i + 6$ are computed as soon as the input operands are ready, these additions are no longer on the critical delay path. Once all the input operands are added using carry-save addition, one more carry-save addition and two 4-bit carry-propagate additions are used to perform the last two corrections and obtain the final sum.

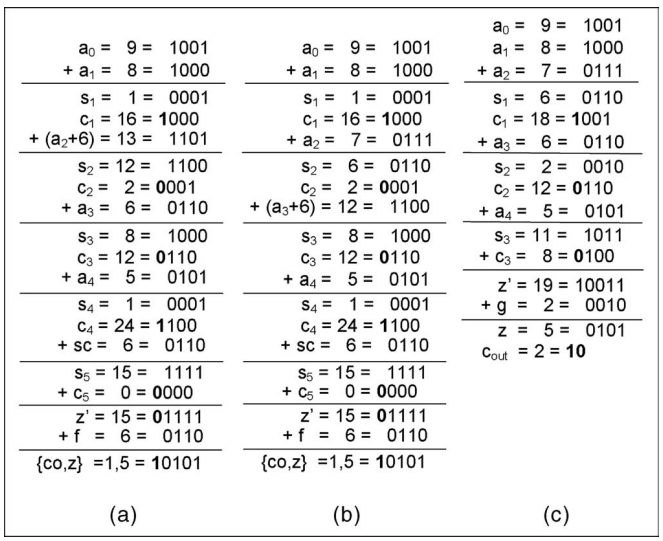

Fig. 4. Examples of 1-digit multioperand decimal addition for $9 + 8 + 7 + 6 + 5 = 35$. (a) Single Correction. (b) Double Correction. (c) Nonspeculative.

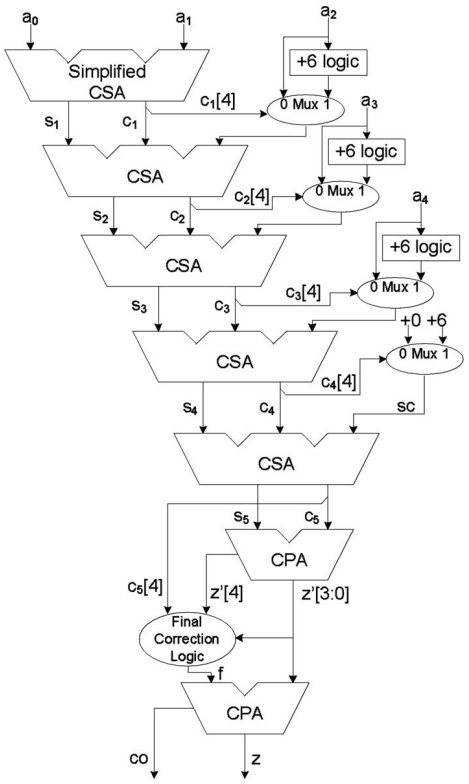

Fig. 5. 1-digit, 5-operand Single Correction Speculation Adder.

A 1-digit, m-operand Single Correction Speculation Adder requires m 4-bit carry-save adders, (m-1) 4-bit 2:1 multiplexers, (m-2) combinational logic blocks to find $a_i + 6$, two 4-bit carry-propagate adders, and one 4-level combinational logic block to produce the final correction, f. Its critical delay path has m carry-save additions, (m-1) 4-bit 2:1 multiplexers, two 4-bit carry-propagate additions, and the logic to implement Table 2, which is a 6-variable Boolean function that is realized in four levels of logic.

3.2 Double Correction Speculation

One improvement that can be made to the Single Correction Speculation Algorithm is to speculate that the first two additions will not need to be corrected. With the Double Correction Speculation Algorithm, shown in Fig. 6, $c_{i-2}[4]$ is used to select whether a_i or $a_i + 6$ is added to s_{i-1} and c_{i-1}. This removes the multiplexers that select between a_i and $a_i + 6$ from the critical path since the correction for a_{i+1} is selected while the carry-save addition of a_i or $a_i + 6$ with s_{i-1} and c_{i-1} is being performed. It also removes the logic needed to produce $a_2 + 6$ since a_2 is always added without a correction value. The only negative consequence of Double Correction Speculation is that determining the speculation correction value, sc, is slightly more complex since two speculative additions need to be corrected, instead of one. The value used for speculation correction is 0, 6, or 12, based on $c_{m-2}[4]$ and $c_{m-1}[4]$, as shown in Table 3. Fig. 4b shows an example of performing the 1-digit BCD addition $9 + 8 + 7 + 6 + 5 = 35$ using the Double Correction Speculation Algorithm.

Fig. 7 shows a block diagram for a 1-digit, 5-operand Double Correction Speculation Adder. A 1-digit, m-operand

```
1) s1 + c1 = a0 + a1;              // using simplified carry-save addition
2) s2 + c2 = s1 + c1 + a2;         // second correction speculation
3) for (i = 3; i < m; i++)         // for each of the remaining operands
       if (ci-2[4]==1)   si + ci = si-1 + ci-1 + (ai + 6);
       else              si + ci = si-1 + ci-1 + ai;
4) sm + cm = sm-1 + cm-1 + sc;     // speculation correction (sc) from Table 3
5) z' = sm + cm;                   // 4-bit carry-propagate addition compressing sum and carry
6) {co, z}  = z' + f;              // f is the final correction from Table 2
```

Fig. 6. Double Correction Speculation Algorithm.

Double Correction Speculation Adder requires m 4-bit carry-save adders, (m-3) 4-bit 2:1 multiplexers, (m-3) combinational logic blocks to find $a_i + 6$, one 4-bit 4:1 multiplexer, two 4-bit carry-propagate adders, and one 4-level combinational logic block to produce the final correction, f. Its critical delay path consists of m carry-save additions, one 4-bit 2:1 multiplexer delay, two 4-bit carry-propagate additions, and 4-levels of logic to implement Table 3. Compared to the Single Correction Speculation Adder, the Double Correction Speculation Adder removes (m-2) 4-bit 2:1 multiplexers from the critical delay path. Speculating beyond two additions does not further decrease delay since the multiplexers are already off of the critical delay path.

3.3 Word-Wide Decimal Addition

The techniques discussed in Sections 3.1 and 3.2 have been described for 1-digit (4-bit) multioperand speculative addition. When adding input operands with multiple digits, the algorithms are performed in the same manner described previously, but $c_i[0]$ is set to the carry-out from the previous less significant carry digit and bit position $c_i[4]$ is passed to the least significant bit of the next more significant carry digit. Thus, an n-digit multioperand decimal adder is composed of n 1-digit multioperand decimal adders that operate in parallel with carries from one digit fed to the next more significant digit. The 4-bit sum digits and 1-bit carry digits produced at the bottom of the 1-digit multioperand adders are fed into a word-wide decimal carry-propagate adder to obtain the final result.

In our designs, the word-wide decimal carry-propagate adder is implemented using a decimal carry-lookahead adder. This adder is similar to the decimal carry-lookahead adder presented in [8], except one of the input operands has digits corresponding to carry bits with values of zero or one. The adder performs word-wide decimal carry-lookahead addition in three steps. First, interdigit carries are generated in parallel. Then, the proper decimal correction values are selected for each digit. Finally, the correction values are added to the 4-bit sum digits.

Fig. 8a and Fig. 8b show examples of word-wide multioperand decimal addition using the Single and Double Correction Speculation Algorithms, respectively. In these examples, capital letters are used to indicate that the operation is performed on operand words with multiple digits. A_i corresponds to input operand A_i after the correction values have been conditionally added. Carries between digits are shown in bold. While adding the input operands, carries out of digit position j become carries into digit position j + 1. The carries out of the most significant digit position can either be summed, if the output operand has more digits than the input operands, or discarded, if the output operand has the same number of digits same as the input operands. When performing the digit carry-propagate additions, $z' = s_{m-1} + c_{m-1}$, each carry-out, $z'[4]$, is concatenated with the final correction, f, of the next more significant digit and then added to $z'[3:0]$. For the word-wide operations, this is denoted as $Z = Z' + (F, Z'[4])$. In the final step, the carry-propagate adder performs $SUM = Z + (R, C0)$, where C0 corresponds to the digit carries from the previous addition and R corresponds to digit corrections produced by the word-wide decimal carry-propagate adder.

TABLE 3
Speculation Correction Selection Table for Step 4

$c_{m-2}[4]$	$c_{m-1}[4]$	sc
0	0	0000 (+0)
0	1	0110 (+6)
1	0	0110 (+6)
1	1	1100 (+12)

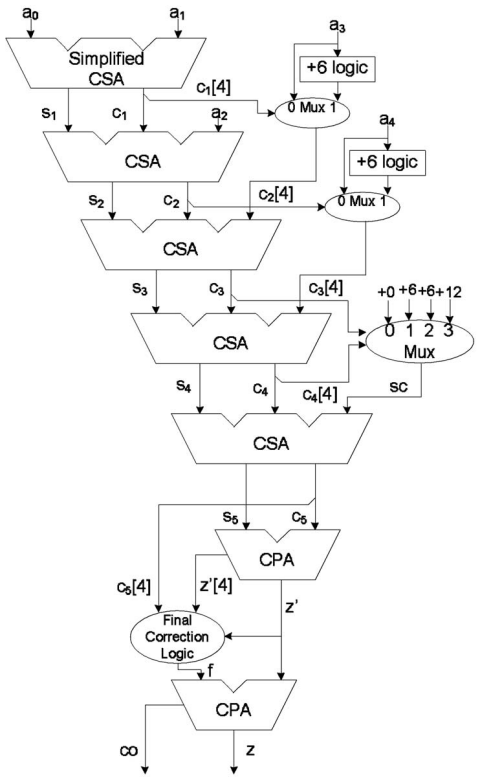

Fig. 7. One-digit, 5-operand Double Correction Speculation Adder.

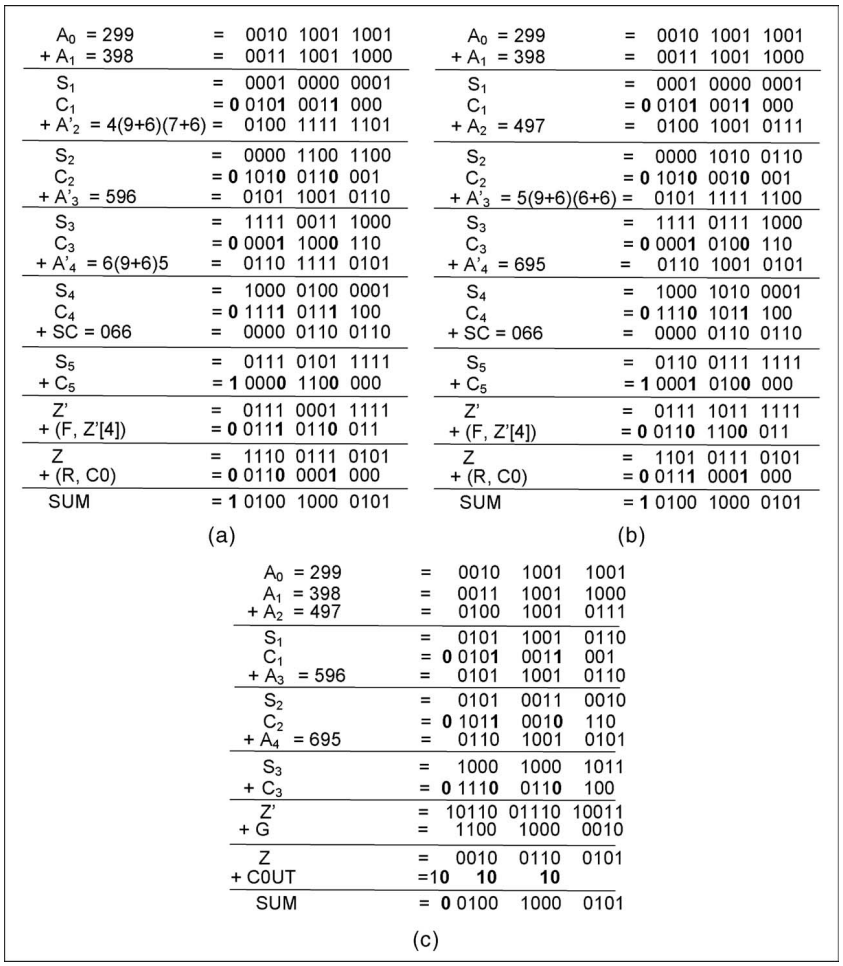

Fig. 8. Examples of multioperand decimal addition for $299 + 398 + 497 + 596 + 695 = 2,485$. (a) Single Correction Speculation. (b) Double Correction Speculation. (c) Nonspeculative.

3.4 Nonspeculative Addition

Nonspeculative Adders sum BCD input operands in a binary carry-save tree, passing carries generated along the way to the next more significant digit. A preliminary binary sum is then produced. These sums and carry-outs from the carry-save adder tree are fed into combinational logic, which produces a decimal sum and carry corrections if needed.

The algorithm for Nonspeculative Addition is shown in Fig. 9. In the algorithm, the input operands are shown as being added serially. In practice, however, a tree of binary carry-save adders is used to add the input operands. Once all the input operands are added, combinational logic is used to determine the sum correction, g, and the carry correction, cout, for the next more significant digit. These values are determined based on the number of carries out of the current digit position, $c_i[4]$ (for $1 \leq i \leq m-2$) and the preliminary sum digit $z'[4:0]$.

With the nonspeculative addition algorithm, the sum and carry correction logic varies based on the number of input operands added. The sum correction, g, is determined by recognizing that 1) a correction of six should be added for each multiple of 10 in the sum of the input digits, 2) each carry-out of the current position, $c_i[4]$ has a weight of 16, and 3) since g is limited to a 4-bit digit, its value is computed modulo 16. Thus, the sum correction is

$$g = \left(\left\lfloor \frac{z'[4:0] + 16 \times \sum_{i=1}^{m-2} c_i[4]}{10} \right\rfloor \times 6 \right) \bmod 16. \quad (4)$$

The carry correction, cout, is determine by recognizing that 1) for each multiple of 10 in the sum of the input digits, a

```
1) s1 + c1 = a0 + a1 + a2;          // using carry-save addition
2) for (i = 2; i < m-1; i++)        // for each of the remaining operands
       si + ci = si-1 + ci-1 + ai+1;
3) z' = sm-2 + cm-2;                // 4-bit carry-propagate addition compressing sum and carry
4) z  = z' + g;                     // g is the sum correction (use Table 4 for m=5)
5) Obtain cout                      // cout is the carry correction (use Table 4 for m=5)
```

Fig. 9. Serial implementation of the Nonspeculative Addition Algorithm.

TABLE 4
Sum and Carry Correction for $m = 5$

$c_1[4] + c_2[4] + c_3[4]$	z'[4:0]	cout	g
0	$0 \le$ z'[4:0] < 10	0	0
0	$10 \le$ z'[4:0] < 20	1	6
0	$20 \le$ z'[4:0] < 30	2	12
0	$30 \le$ z'[4:0] < 32	3	2
1	$0 \le$ z'[4:0] < 4	0	6
1	$4 \le$ z'[4:0] < 14	1	12
1	$14 \le$ z'[4:0] < 24	2	2
1	$24 \le$ z'[4:0] < 32	3	8
2	$0 \le$ z'[4:0] < 8	1	2
2	$8 \le$ z'[4:0] < 16	2	8

carry-out of the current digit position should be generated and 2) the carries, $c_i[4]$, that have already been passed to the next more significant digit do not need to be produced by the carry correction logic. Thus, the carry correction is:

$$cout = \left\lfloor \frac{z'[4:0] + 16 \times \sum_{i=1}^{m-2} c_i[4]}{10} \right\rfloor - \sum_{i=1}^{m-2} c_i[4]. \quad (5)$$

Because of how cout is determined, $z'[4]$ and $z[4]$ are discarded, instead of being passed to the next more significant digit, which simplifies the overall hardware design.

For a given number of input operands, m, efficient combinational logic can be designed to produce g and cout. Table 4 shows how g and cout are determined based on the sum of the intermediate digit carries, $c_i[4]$ (for $1 \le i \le m-2$) and the preliminary sum, $z'[4:0]$, when $m = 5$. For example, when $c_1[4] + c_2[4] + c_3[4] = 1$ and $14 \le z'[4:0] < 24$, the sum of the input digits is $30 \le a_0 + a_1 + a_2 + a_3 + a_4 < 40$ since $c_i[4]$ has a weight of 16. Thus, the sum correction is $g = (3 \times 6) \bmod 16 = 2$ and the carry correction is $cout = 3 - 1 = 2$. The same basic approach is used for other values of m, but the sum and carry correction logic becomes more complex as m increases.

Fig. 4c shows an example of performing the 1-digit BCD addition $9 + 8 + 7 + 6 + 5 = 35$ using the Nonspeculative Addition Algorithm. In this example, $c_1[4] + c_2[4] + c_3[4] = 1$ and $z'[4:0] = 19$. Based on Table 4 or (4) and (5), this gives $cout = 2$ and $g = 2$. Adding g to $z'[3:0]$ gives 5 and adding cout to $c_1[4] + c_2[4] + c_3[4]$ gives 3, so the correct result of 35 is produced. As noted previously, $z'[4]$ is discarded, which simplifies the carry correction logic.

Fig. 10 shows a block diagram for a 1-digit, 5-operand Nonspeculative Adder. The five BCD operand digits are added using a binary carry-save tree and a 4-bit CPA. The result is a 5-bit binary sum, $z'[4:0]$, and three intermediate carry-outs, $c_1[4]$, $c_2[4]$, and $c_3[4]$. The 5-bit binary sum and the three carry-outs are fed into sum and carry correction logic to produce g and cout, which is passed to the next more significant digit. The sum correction, g, and the lower four bits of the binary sum, $z'[3:0]$, are passed through a 1-digit CPA to produce the correct BCD sum, z.

A 1-digit, m-operand Nonspeculative Adder requires (m-2) 4-bit carry-save adders, one 4-bit carry-propagate adder, one 5-level combinational logic block to generate the carry-out and correction digits (for up to 16 input operands), and one 3-bit carry-propagate adder to add the correction digit to the binary sum. Its critical delay path consists of roughly $\lfloor \log_{3/2}(m-1) \rfloor$ carry-save additions, one 4-bit carry-propagate addition, one 5-level logic block, and one 3-bit carry-propagate addition. Unlike the Correction Speculation Adders, which use an array of binary carry-save adders and have a linear delay, the Nonspeculative Adders use a tree of binary carry-save adders and have logarithmic delay.

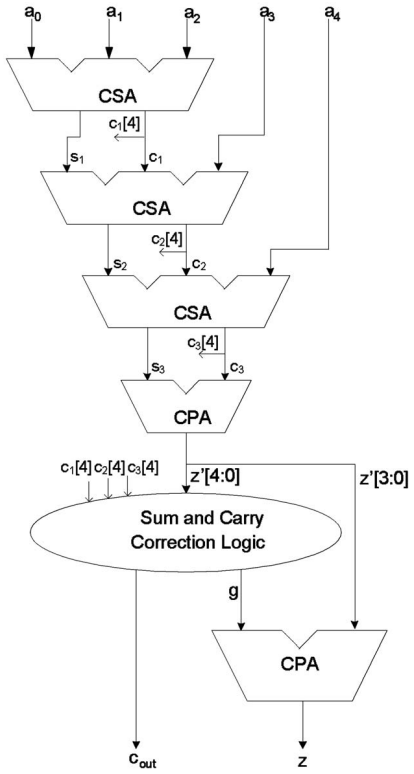

Fig. 10. One-digit, 5-operand Nonspeculative Adder.

The word-wide BCD adder for the Nonspeculative Adders uses decimal carry-lookahead logic, similar to that described in [8]. The addition is done using a variation of direct decimal addition [8] in which each 1-digit adder takes a sum and carry digit and produces digit propagate and generate signals. The digit propagate and generate signals are then sent to carry-lookahead logic, which is used to compute digit carries in parallel. Finally, the digit carries and additional carry-lookahead logic within each digit are used to quickly produce the sum digits.

An example of word-wide Nonspeculative Addition is shown in Fig. 8c. In this example, the input operands are added using three word-wide carry-save additions and parallel digit-wide carry-propagate additions to produce Z'. For each digit, $z'[4:0]$ and $c_4[i]$ ($1 \le i \le m-2$) are examined to determine the g and cout. G is added to Z' using parallel digit-wide carry-propagate additions to produce Z and then COUT is added to Z using decimal carry-lookahead addition [8] to produce SUM.

3.5 Binary Multioperand Adders

For comparison, binary multioperand carry-save adders are built to evaluate the additional cost of performing multioperand decimal addition. One set of binary multioperand

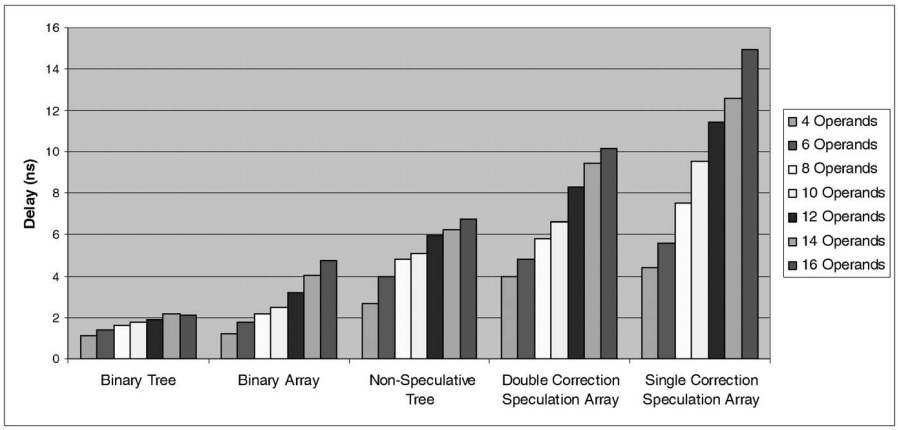

Fig. 11. Delay for 4-bit multioperand adders.

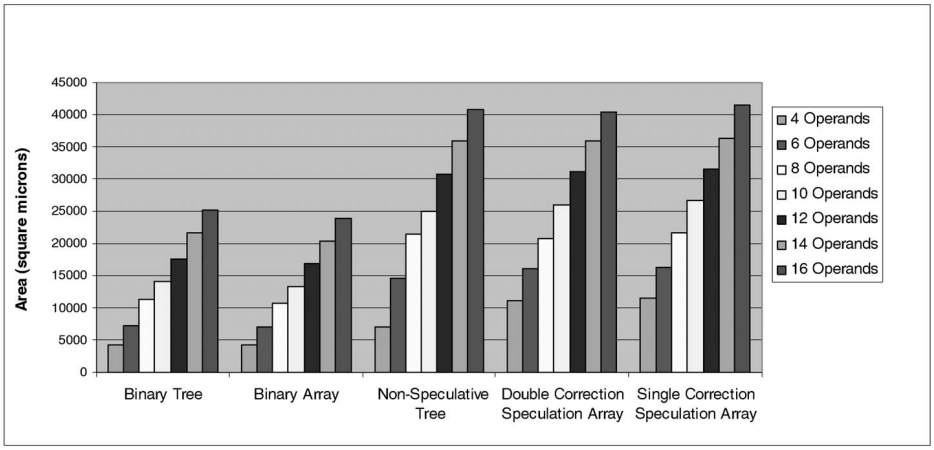

Fig. 12. Area for 4-bit multioperand adders.

adders is designed to be similar to the Correction Speculation Adders and contains a linear array of binary carry-save adders. The other set is designed to be similar to the Nonspeculative Adders and uses a tree of binary carry-save adders. Both types of binary multioperand adders use the same word-wide carry-propagate adder. In the word-wide carry-propagate adder, two levels of carry-lookahead logic are implemented. The first level produces group generate and propagate signals for 4-bit blocks. The second level uses the group generate and propagate signals to obtain the carries into each 4-bit block.

4 Synthesis Results

Decimal and binary multioperand adders were modeled in Verilog and simulated extensively to ensure correct functionality. They were then synthesized using Synopsys Design Compiler and the LSI Logic's lcbg11p 0.18 micron CMOS standard cell library. When performing synthesis, the designs were optimized for area. Four-bit (1-digit) and 32-bit (8-digit) multioperand adders were constructed for each of the techniques proposed, including the two types of binary multioperand adders discussed in Section 3.5. Each 32-bit multioperand decimal adder is constructed from eight 1-digit multioperand adders, followed by a word-wide decimal carry-lookahead adder. Each 32-bit binary multioperand adder is constructed using a linear array or logarithmic tree of carry-save adders, followed by a word-wide binary carry-lookahead adder. The number of input operands for the adders constructed ranges from four to 16 operands.

The delay and area for the 4-bit multioperand adders are shown in Figs. 11 and 12, respectively. The delay and area for the 32-bit adders are shown in Figs. 13 and 14, respectively. Similar conclusions can be reached using either the 4-bit or the 32-bit multioperand adder results. The 32-bit multioperand adder results, which show the overall area and delay due to processing multiple digits and performing word-wide carry-lookahead addition, are discussed throughout the rest of this section.

For all speculative multoperand adders and also the binary array multioperand adders, the delay increases linearly with the number of input operands. The difference in delay between Double and Single Correction Speculation Adders grows with more input operands because the multiplexer delays to select a_i or $a_i + 6$ are hidden in the Double Correction Speculation Adders.

The Nonspeculative Adders have lower delays than all of the Speculative Adders. The biggest advantage of the Nonspeculative Adders is that their delay grows logarithmically, rather than linearly, since operands are added using a tree of binary carry-save adders. Their logarithmic

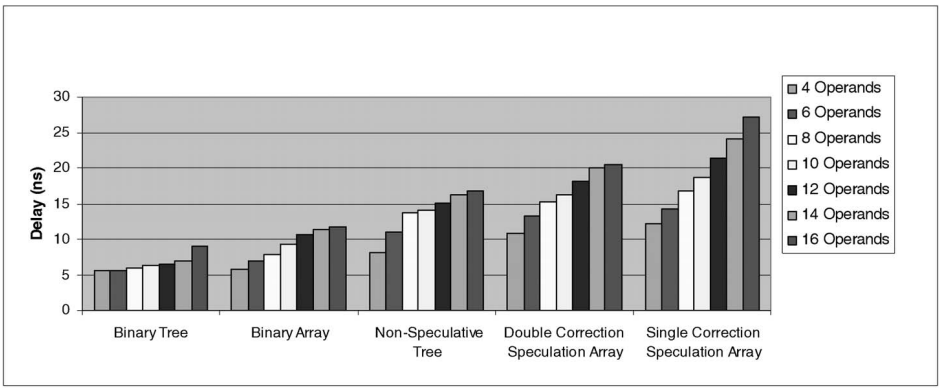

Fig. 13. Delay for 32-bit multioperand adders.

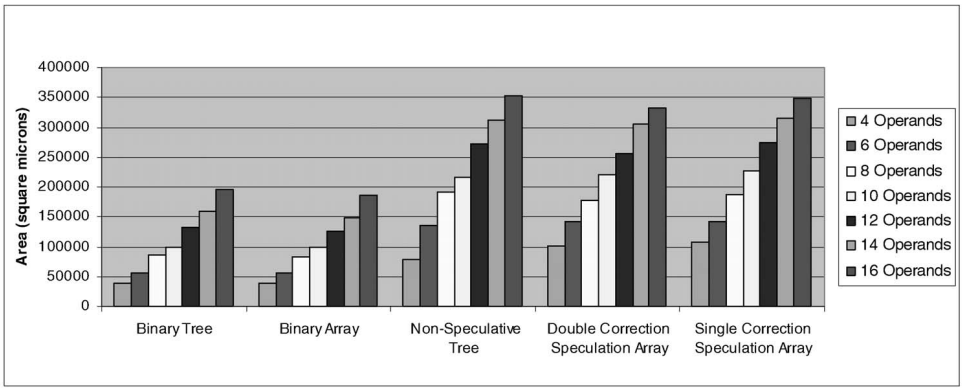

Fig. 14. Area for 32-bit multioperand adders.

delay is particularly useful when a large number of input operands are added. The area for each of the decimal adders is similar. On average, the Double Correction Speculation Adders have slightly less area than the other decimal adders. The Nonspeculative Adders can also be implemented using (4:2) compressors to further improve performance and regularity [12].

Although the Nonspeculative Adders have a significant delay advantage over the Speculative Adders for the selected standard cell libraries, there are situations in which the Speculative Adders may have some distinct advantages. First, the designs of the Speculative Adders are more regular, which made lead to advantages in full-custom, deep-submicron design. Second, the final correction logic for the Speculative Adders is independent of the number of input operands. This may lead to advantages when designing iterative multioperand adders in which the number of input operands can vary since the same final correction logic can be used regardless of the number of input operands.

The areas and delays for binary adders are shown for comparison. The cost of performing multioperand decimal addition versus multioperand binary addition is calculated by comparing the Nonspeculative Adders, which have the smallest delay and small overall area, to the Binary Tree Adders. The Nonspeculative Adders have 1.44 to 2.34 times more delay and 1.61 to 2.03 times more area than the Binary Tree Adders.

5 Related Work on Decimal Addition

Many techniques have been developed to speed up the process of decimal addition [7], [8], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23]. Since these techniques were designed for either adding two decimal operands or for adding a small number of partial products in decimal multipliers, they are not as well suited for multioperand decimal addition as the techniques presented in this paper. In this section, we summarize the previous techniques for decimal addition that can be more efficiently adopted to perform multioperand decimal addition and compare them to our proposed techniques. Further details on the previous techniques are provided in [8], [14], [15], [16].

One efficient technique for two-operand decimal addition is *direct decimal addition* [8]. With this technique, combinational logic first produces digit generate and propagate signals for each pair of BCD digits. These signals are similar to the group generate and propagate signals used in binary carry-lookahead adders [5], but are modified to correctly handle decimal operands. The digit propagate and generate signals are then sent to carry-lookahead logic, which is used to compute digit carries in parallel. Finally, the digit carries and additional carry-lookahead logic within each digit are used to quickly produce the sum digits, without the need for a final correction. Using this approach repetitively to perform multioperand decimal addition has far more delay than the techniques presented in this paper, since adding m input operands requires (m-1) word-wide carry-propagate additions.

Erle and Schulte proposed a different technique for decimal addition for use in an iterative decimal multiplier [14]. With this technique, a variant of direct decimal addition is used to produce intermediate results in a decimal carry-save format. As introduced in [14], a decimal (3:2) counter accepts as inputs two 4-bit BCD digits and a 1-bit carry and outputs a 4-bit BCD sum digit and a 1-bit carry. Similarly, a decimal (4:2) compressor accepts as inputs two 4-bit BCD digits and two 1-bit carries and outputs a 4-bit BCD sum digit and a 1-bit carry. To perform multioperand addition, the decimal (4:2) compressor presented in [14] can be modified to take four BCD digits and produce a 4-bit sum digit and a 2-bit carry digit since the largest sum of four BCD digits is 36. A tree of modified decimal (4:2) compressors can then be used to add the input operands, using techniques similar to those employed by binary (4:2) compressors [10], to produce the result in decimal carry-save format. A word-wide decimal carry-propagate adder can then be used to add the resulting sum and carry digits. This approach, however, is slower and requires more hardware than the Nonspeculative Adders presented in this paper since each 1-digit modified decimal (4:2) compressor requires two binary carry-save additions, a 4-bit binary carry-propagate addition, and decimal sum correction.

Another approach, proposed by Ohtsuki et al., uses decimal carry-save addition in a decimal multiplier to iteratively accumulate partial products [15]. With their approach, a correction value of six is added to each digit of the first partial product using a binary carry-save adder. For each partial product added, the 4-bit sum and 4-bit carry outputs of the binary carry-save adder are examined, using fairly complex logic, to see if a correction value of zero, six, or 12 should be added to the sum digit using a second carry-save addition. After all of the partial products have been added, a word-wide carry-propagate addition is performed. Each digit with a carry-out of zero is then corrected by subtracting six from the digit. The technique presented in [15] is similar to the speculative correction techniques described in Sections 3.1-3.3, with partial products being analogous to input operands. One advantage of our speculative correction techniques is that our correction determination logic is much simpler since only the most significant bit of the carry digit is examined. Another advantage is that the correction value is added to the input operands via combinational logic, whereas, in [15], the correction is added to the new sum, which is on the critical path of their circuit. With the technique presented in [15], each input operand added requires two binary carry-save additions and fairly complex sum digit correction logic. Although the final correction step used in [15], is simpler than the final correction step in our techniques, our techniques are still superior since the iterative portion is faster and requires less hardware. Furthermore, our Nonspeculative technique has logarithmic delay, while the technique presented in [15] has linear delay.

Shirazi et al. propose a technique for constant time decimal addition, called Redundant Binary Coded Decimal (RBCD), in [16], [17]. When using RBCD to add multiple BCD operands, the BCD operands are first converted to an RBCD representation and then added. Both conversion from BCD to RBCD and RBCD addition are done in constant time because carry propagation is eliminated. When the multioperand addition is complete, the RBCD sum is converted back to BCD using a circuit that is similar to a word-wide carry-propagate adder. One advantage to using our techniques over RBCD is that no conversion to or from RBCD is required. Our techniques also have the advantage in that they only have one carry-save addition on the critical path for each input operand. In comparison, the RBCD adder has two 4-bit carry-propagate adders and two PLA accesses for each digit addition performed. Our techniques need two 4-bit carry-propagate additions only after all of the operands have been added. Thus, our techniques are faster and use less area, than the techniques presented in [16], [17].

6 Conclusions

In this paper, we have proposed three algorithms for multioperand decimal addition. The first two algorithms, which have linear delay, speculate BCD corrections and correct results based on carries from previous additions. The third algorithm, which has logarithmic delay, adds the input operands using a tree of carry-save tree adders and produces a binary sum. Combinational logic is then used to correct the binary sum and produce the proper decimal carry. Our studies show that, compared to the Speculative Adders, the Nonspeculative Adders have less delay and roughly the same area. Compared to previous techniques for decimal addition, our techniques provide a significant area and speed advantage when multiple decimal input operands are added.

Acknowledgments

This research was supported in part by an IBM Faculty Award. The authors are grateful to the anonymous reviewers for their excellent advice on revising the paper.

References

[1] M. Hill, N. Jouppi, and G. Sohi, *Readings in Computer Architecture.* San Francisco: Morgan Kaufmann, 2000.
[2] M.F. Cowlishaw, "Decimal Floating-Point: Algorism for Computers," *Proc. 16th IEEE Symp. Computer Arithmetic,* pp. 104-111, June 2003.
[3] *Draft IEEE Standard for Floating-Point Arithmetic.* New York: IEEE, Inc., 2004, http://754r.ucbtest.org/drafts.
[4] *IEEE Standard for Binary Floating-Point Arithmetic.* New York: IEEE Inc., 1985.
[5] P. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs.* New York: Oxford Univ. Press, 2000.
[6] R.D. Kenney and M.J. Schulte, "Multioperand Decimal Addition," *Proc. IEEE CS Ann. Symp. VLSI,* pp. 251-253, Feb. 2004.
[7] R.K. Richards, *Arithmetic Operations in Digital Computers.* D. Van Nostrand Company, Inc., 1955.
[8] M. Schmookler and A. Weinberger, "High Speed Decimal Addition," *IEEE Trans. Computers,* vol. 20, no. 8, pp. 862-867, Aug. 1971.
[9] C.-H. Yeh and B. Parhami, "Efficient Pipelined Multi-Operand Adders with High Throughput and Low Latency: Designs and Applications," *Conf. Record 30th Asilomar Conf. Signals, Systems and Computers,* vol. 2, pp. 894-898, Nov. 1996.

[10] P. Kornerup, "Reviewing 4-to-2 Adders for Multi-Operand Addition," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors*, pp. 218-229, July 2002.
[11] C.S. Wallace, "Suggestion for a Fast Multiplier," *IEEE Trans. Electronic Computers*, vol. 13, pp. 14-17, 1964.
[12] M.T. Santoro and M.A. Horowitz, "SPIM: A Pipelined 64x64 Bit Iterative Multiplier," *IEEE J. Solid-State Circuits*, vol. 24, pp. 487-494, 1989.
[13] H. Schmid, *Decimal Computation.* John Wiley & Sons, 1974.
[14] M.A. Erle and M.J. Schulte, "Decimal Multiplication via Carry-Save Addition," *Proc. IEEE 14th Int'l Conf. Application-Specific Systems, Architectures, and Processors*, pp. 348-358, June 2003.
[15] T. Ohtsuki et al., "Apparatus for Decimal Multiplication," US Patent #4,677,583, June 1987.
[16] B. Shirazi, D.Y. Yun, and C.N. Zhang, "RBCD: Redundant Binary Coded Decimal Adder," *IEE Proc.—Part E*, vol. 136, no. 2, Mar. 1989.
[17] B. Shirazi, D.Y. Yun, and C.N. Zhang, "VLSI Designs for Redundant Binary-Coded Decimal Addition," *Proc. Seventh Ann. Int'l Conf. Computers and Comm.*, pp. 52-56, Mar. 1988.
[18] W. Bultmann, W. Haller, H. Wetter, and A. Worner Alexander, "Binary and Decimal Adder Unit," US Patent #6,292,819, Sept. 2001.
[19] J.R. Eaton and K. Hughes, "Decimal Arithmetic Apparatus and Method," US Patent #5,745,399, Apr. 1998.
[20] S. InSeok, "High-Speed Binary and Decimal Arithmetic Logic Unit," US Patent #4,866,656, Sept. 1989.
[21] S. Singh, "High-Speed Radix 100 Parallel Adder," US Patent #6,546,411, Apr. 2003.
[22] W. Haller, U. Krauch, T. Ludwig, and H. Wetter, "Combined Binary/Decimal Adder Unit," US Patent #6,546,411, July 1999.
[23] M.J. Adiletta and V.C. Lamere, "BCD Adder Circuit," US Patent #4,805,131, Feb. 1989.
[24] J.L. Anderson, "Binary or BCD Adder with Precorrected Result," US Patent #4,172,288, Oct. 1979.

Robert D. Kenney received the bachelor's degree in computer engineering from the University of Wisconsin-Madison in 2002. He received the master's degree in electrical engineering from the University of Wisconsin-Madison in 2004, where, under the direction of Dr. Schulte, he focused his research efforts on decimal arithmetic. He is currently employed at IBM designing test methodologies for the multicore Cell Processor.

Michael J. Schulte received the BS degree in electrical engineering from the University of Wisconsin-Madison in 1991, and the MS and PhD degrees in electrical engineering from the University of Texas at Austin in 1992 and 1996, respectively. From 1996 to 2002, he was an assistant and associate professor at Lehigh University, where he directed the Computer Architecture and Arithmetic Research Laboratory. In 1997, he received a US National Science Foundation CAREER Award to research hardware support for accurate and reliable numerical computations. He is currently an assistant 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, computer arithmetic, and wireless security. He is a senior member of the IEEE and the IEEE Computer Society and an associate editor for the *IEEE Transactions on Computers* and the *Journal of VLSI Signal Processing*.