Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Fully redundant decimal addition and subtraction using stored-unibit encoding

Amir Kaivani [a], Ghassem Jaberipur [a,b,*]

[a] Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran
[b] School of Computer Science, Institute for Studies in Theoretical Physics and Mathematics (IPM), Tehran, Iran

ARTICLE INFO

Article history:
Received 19 February 2008
Received in revised form
30 June 2008
Accepted 28 April 2009

Keywords:
Decimal computer arithmetic
Redundant number systems
Fully redundant addition
Semi-redundant addition
Stored-unibit encoding

ABSTRACT

Decimal computer arithmetic is experiencing a revived popularity, and there is quest for high-performance decimal hardware units. Successful experiences on binary computer arithmetic may find grounds in decimal arithmetic. For example, the traditional fully redundant (i.e., the result and both of the operands are represented in a redundant format) and semi-redundant (i.e., the result and only one of the operands are redundant) binary addition schemes have influenced the design and implementation of similar decimal arithmetic units. However, special comparison and correction steps are required when decimal arithmetic algorithms are implemented on binary hardware. To circumvent these difficulties, alternative encodings of decimal digits and a variety of decimal arithmetic algorithms have been examined by many researchers over decades. In this paper we offer a new redundant decimal digit set $[-8, 9]$ and a fully redundant addition/subtraction scheme. The proposed digit set, faithfully encoded as a mix of posibits, negabits, and unibits, is shown to obviate the need for any compare-to-9 operations and leads to minimal penalty subtraction using the addition circuitry. Moreover, conversion from the standard BCD encoding to the proposed stored-unibit encoding is possible with the latency of one logic level. However, the reverse conversion, like any other redundant to nonredundant conversion, involves carry propagation.

1. Introduction

The supercomputing needs in today's commercial, financial, scientific, and internet-based applications [2] have led the industry towards the commercialization of digital processors with embedded decimal arithmetic unit such as IBM POWER6 processor [25]. Also, specifications for decimal number representation and arithmetic have been incorporated in the IEEE P754 standard for floating point arithmetic [8].

Redundant number systems and the related arithmetic operations have been used in numerous implementations of digital arithmetic units [7]. Within a composite arithmetic operation it is common to represent the intermediate results in a redundant format (e.g., carry-save representation of accumulated partial products in multiplication). Use of redundant representation allows for carry-free addition and subtraction, where the latency is independent of the number of digits. Carry-save [18] addition scheme is a representative example of semi-redundant operations, where only one of the operands is redundant. However, in fully redundant operations, such as in signed-digit adders [1], the result and both of the operands are represented in a redundant format. Likewise, in decimal computer arithmetic, semi-redundant and fully redundant addition and subtraction schemes have been a topic of interest. For example, sequential decimal multiplication in [14], multi-operand decimal addition in [3,15], parallel decimal multiplication in [16,29], and decimal division in [17,30] use decimal carry-save addition. However, fully redundant decimal addition and/or subtraction have been addressed as independent operations [28,26,19], and as a building block for sequential decimal multiplication [5] and division [20]. Nevertheless no additional efficiency, due to fully redundant (VS semi-redundant) add/subtract, is claimed by the latter two contributions.

Fully redundant adders, if used in parallel multiplication, lead to VLSI-friendly recursive partial product tree. For example, consider the binary signed-digit adders used for partial product reduction in binary multipliers (e.g., [4]). The only use of fully redundant decimal adders, within composite operations, that we have come across is in the sequential decimal multiplier of [5] and divider of [20]. However, there are specialized applications such as redundant CORDIC arithmetic that intrinsically require fully redundant addition/subtraction [4]. Moreover, one might think of whole computation applications where several fully redundant arithmetic operations may take place before a result is stored in

* Corresponding author.
E-mail addresses: A_Kaivani@sbu.ac.ir (A. Kaivani), Jaberipur@sbu.ac.ir (G. Jaberipur).

0167-9260/$ - see front matter © 2009 Published by Elsevier B.V.
doi:10.1016/j.vlsi.2009.04.001

memory (e.g., [6]). In such computing hardware structures it is desired to keep the number of data paths and interconnections between arithmetic units as small as possible. Therefore, redundant decimal encodings with less number of bits per digit may prove more useful if other advantages are not dramatically lost. We are thus motivated to explore more efficient encodings for redundant decimal digits and fully redundant decimal addition/subtraction algorithms.

The rest of this paper is organized as follows. A brief background on decimal adders and subtracters is offered in Section 2. Previous works, on fully redundant decimal adders, are addressed in Section 3. In Section 4 we reproduce a description of two-valued-digits (Twit) and weighted-twit-set (WTS) encoding of redundant digit sets [11], and present a particular WTS encoding of decimal digits. The proposed fully redundant decimal adder of this work is described in Section 5, and subsequently in Section 6, we explain how it can be adapted to perform decimal subtraction with minimal additional latency. Conversion to and from conventional decimal representation is taken up in Section 7. A comprehensive comparison of our results with the previous fully redundant add/subtract circuits, based on Logical effort [27] analysis, is presented in Section 8. Finally our conclusions are drawn in Section 9.

2. Background

The binary-coded-decimal (BCD) encoding is the dominant representation for decimal digits. Each BCD digit is represented by 4 bits with power-of-two weights 8, 4, 2, and 1. However, radix 10 is not a power of two and this imposes particular difficulties on implementation of decimal adders using two-valued logic and binary arithmetic cells:

(a) *Over-9 detection*: It is desirable to use standard 4-bit binary adders to add two BCD digits. However, no carry-out is generated for digit-sum values in the range [10, 15]. Therefore, to decide on decimal carry-out, a comparison with 9 is generally required.
(b) +6 *Correction*: Interpreting the carry-out of the 4-bit addition as a decimal carry may impose a +6 correction operation on the sum digit.
(c) *Unified add/subtract logic*: This is desirable in decimal arithmetic units as is common in binary arithmetic.

Conventional BCD addition schemes [24] implement the digit equation $x^i+y^i+c_i=10c_{i+1}+s^i$, where x^i and y^i are input BCD digits in position i. These addition schemes use one 4-bit binary adder per decimal position to produce an interim 5-bit sum w^i in [0, 19] = [0, 9]+[0, 9]+[0, 1], where the last interval relates to the decimal carry-in c_i, coming from the next less-significant digit position. The interim sum w^i is decomposed to a sum BCD-digit s^i and a decimal carry-out c_{i+1}. Therefore some mechanism for over-9 detection is required to determine c_{i+1}. The whole process for k-digit BCD numbers may be described as Algorithm 1, as follows:

Algorithm 1 (*Conventional BCD addition*).
Inputs: k-digit BCD numbers $X=x^{k-1},\ldots,x^0$ and $Y=y^{k-1},\ldots,y^0$, where $x^i=x_3^ix_2^ix_1^ix_0^i$ and $y^i=y_3^iy_2^iy_1^iy_0^i$, for $0\le i\le k-1$.
Output: $S=s^{k-1},\ldots,s^0=X+Y$, and the overflow signal $v=c_k$.

```
Set $c_0=0$ and for $i=0$ to $k-1$ do
 I. Compute the interim binary digit-sum $w^i=w_4^iw_3^iw_2{}^iw_1^iw_0^i=x^i+y^i+c_i$.
 II. If $w^i>9$ then $s^i=w_3^iw_2^iw_1^iw_0^i$+0110, and $c_{i+1}=1$ else
     $s^i=w_3^iw_2^iw_1^iw_0^i$ and $c_{i+1}=0$.
```

There are alternative encodings for decimal digits that overcome some of the difficulties listed above. For example, in the Excess-3 encoding [23], a decimal digit d in [0, 9] is encoded as a 4-bit binary representation of d+3. The main benefit is that the carry-out of a standard 4-bit adder, receiving two Excess-3 digits, can directly serve as the decimal carry, thus obviating the need for comparison with 9.

As another example, consider the 4-bit decimal encoding with weights {8,−4,−2,1} [31], where no bit combination can assume a value greater than 9, hence obviating the need for the comparison operation. However, the digit adder is a specialized one with extra complexity and more latency than a conventional 4-bit adder.

Using the addition circuitry for subtraction is common in binary arithmetic, where the subtrahend is negated and then added to the minuend. For example, in the standard two's complement adder/subtractor, the penalty for subtraction is only one XOR gate per binary position. In decimal subtraction, however, the ten's complement operation is more involved. Algorithm 2 describes the details.

Algorithm 2 (*Decimal subtraction*).
Inputs: k-digit ten's complement decimal numbers $X=s_xx^{k-1},\ldots,x^0$ and $Y=s_yy^{k-1},\ldots,y^0$.
Output: $D=s_dd^{k-1},\ldots,d^0=X-Y$, and overflow flag v.
Notation: x^i, y^i, and d^i $(0\le i\le k-1)$ are unsigned decimal digits, and s_x, s_y, and s_d are (-10^k)-weighted sign bits.

```
I. Compute the nine's complement $\overline{y^i}=9-y^i$of each digit $y^i$, for
   $0\le i\le k-1$.
II. Perform the enforced carry addition $c_kd^{k-1},\ldots,d^0=x^{k-1},\ldots,$
   $x^0+\overline{y^{k-1}}\ldots\overline{y^0}+1$.
III. Derive the sign bit of the result $s_d=\overline{s_x\oplus s_y\oplus c_k}$, and the
   overflow flag $v=c_k\overline{s_x}s_y+\overline{c_k}s_x\overline{s_y}$.
```

To reduce the overhead of performing decimal subtraction using addition circuitry, it is desirable to minimize the latency of Step I. Two levels of logic are required for the BCD nine's complement operation [24], while the same is achieved, in Excess-3, by bit-wise inversion. Unfortunately, however, the latter requires a complex post-correction step per digit, which involves a conditional add-±3 operation depending on the value of carry-out. There is also the 4-2-2-1 encoding of decimal digits [29], where nine's complementation is possible via bit-wise inversion. However, we have not encountered any decimal adder/subtractor based on such encoding. Finally the nine's complementation of {8,−4,−2,1}-encoded decimal digits is possible, but the specialized adder cells provided in [31] are not 4-bit binary adders and cannot accept nine's complemented digits.

3. Redundant-digit decimal arithmetic

The main benefit of redundant number systems is the possibility of carry-free addition, where addition is performed in a small constant time independent of the number of digits in the operands [21]. Algorithm 3 (reproduced from [12] for ease of reference) describes the steps of carry-free addition. Conversion of a number, represented in a conventional nonredundant encoding (e.g., two's complement, or BCD), to its equivalent representation in a redundant encoding is also a carry-free operation. However, the reverse conversion requires wordwide carry propagation, where the latter may be amortized over per-operation savings compounded by many redundant operations.

Algorithm 3 (*Conventional carry-free radix-r addition*).
Inputs: k-digit radix-r numbers $X = x^{k-1},\ldots,x^0$ and $Y = y^{k-1},\ldots,y^0$.
Output: $S = s^{k-1},\ldots,s^0 = X+Y$.
Notation: x^i, y^i, and s^i $(0 \le i \le k-1)$ are radix-r digits in $[\alpha,\beta]$.
Concurrently perform the following digit operations for all radix-r positions $(0 \le i \le k-1)$:

```
I. Compute the position sum $p^i = x^i + y^i$.
II. Derive the interim sum digit $w^i$ and transfer digit $t^{i+1}$ satisfying $w^i = p^i - rt^{i+1}$.
III. Form the final sum digit $s^i = w^i + t^i$.
```

The interim sum and transfer digit in Step II are chosen such that the computed sum digit in Step III falls within $[\alpha, \beta]$. The performance efficiency of this algorithm depends, to some extent, on the encoding of redundant digits. For example the stored transfer representation of redundant numbers obviates the need for Step III [11]. To take advantage of this improvement, we propose a stored transfer representation of decimal digits in the next section.

The digit set of a redundant number system should have more members than the radix of the representation. Thus the digits of a radix-2^h redundant number system must be represented with more than h bits. When the radix is not a power of two, however, it may be possible to avoid using any extra bits to encode the digit. For example, Shirazi et al. represent the digit values of a radix-10 digit set $[-7, 7]$ as 4-bit two's complement numbers. This digit set is redundant since the number of digits (i.e., $7+7+1 = 15$) exceeds the radix (i.e., 10). Note that the number of bits in the encoding is equal to the minimum required for any nonredundant decimal encoding (i.e., $\lceil \log 10 \rceil = 4$). As another example, consider the maximally redundant decimal digit set $[-9, 9]$ represented by a positive and a negative equally weighted BCD digits [19]. Given that the latter digit set may be represented by only 5 bits (i.e., $\lceil \log 19 \rceil$), the 8-bit encoding of [19] needs to be carefully examined for possible advantages (see Section 8.3). Besides the latter signed-digit sets, there are instances of redundant decimal digit sets with no negative value. These two types of redundant decimal digit sets are further distinguished below:

(a) *Nonnegative digit sets*: The designs of decimal multipliers by [16,29] imply the use of digit sets (0, 10) and (0, 18), respectively. However, there are ten's complement signed multiples in the first level of the partial product tree. These signed multiples are generated because the 8- and 9-multiples are represented as $10X$–$2X$ and $10X$–X, respectively, where X is the multiplicand. Only one of the operands in the first reduction level, which is indeed nonredundant, can be negative. Therefore, after this level, the reduced partial products are all positive. However, in fully redundant applications with nonnegative digit sets, either sign-magnitude or radix-complement encoding is required, where some obstacles get in the way of efficient processing. For example, consider the swapping of the operands and post-complementation in the sign-magnitude redundant digit floating point addition in [6]. The radix-complement practice may be exemplified by the double 4-2-2-1 decimal encoding in [29], where ten's complementation leads to two equally weighted sign bits that is difficult to handle.

(b) *Signed-digit* (*SD*) *sets*: All the fully redundant SD decimal addition schemes that we have come across have used balanced digit sets (i.e., $[-6, 6]$ in [28], $[-7, 7]$ in [26], and $[-9, 9]$ in [19]) with no separate sign bit for the whole number. Such sign-embedded encodings lead to almost similar treatment of addition and subtraction.

4. Stored transfer representation of decimal digits

The straightforward implementation of Algorithm 3 involves two digit-wide additions in Steps I and III. The latter may be postponed until the next addition, thus saving time and energy. This intuition has led to the introduction of stored transfer representation of redundant digit set in [9], extended in [10] as weighted-bit-set (WBS) encoding, further extended in [11] as weighted-twit-set (WTS) encoding, and theoretically supported in [12].

A unified adder/subtractor based on stored-unibit-transfer (SUT) encoding demonstrated some advantages, particularly in terms of speed [11]. In this paper we apply a similar approach to a redundant decimal adder/subtractor and find that similar advantages can be obtained. We reproduce three definitions on twit, unibit, and WTS encoding from [11], as prerequisites for defining the SUT decimal digit set.

Definition 1 (*Twit*). A twit (i.e., two-valued digit) denoted as $\{\lambda, \lambda+\gamma\}$ is logically represented by a bit. However, it represents two arithmetic values λ and $\lambda+\gamma$, where λ is the twit's lower value and γ $(\gamma > 0)$ is the gap between λ and the higher value $\lambda+\gamma$. For example, a posibit $\{0, 1\}$ has $\lambda = 0$, and $\gamma = 1$, and a negabit $\{-1, 0\}$ is a twit with $\lambda = -1$, and $\gamma = 1$. Table 1 describes symbolic and dot notations of three useful twits.

The encoding for negabit, as a twit, is exactly the opposite of what is conventionally used, for example, for the most significant bit of a two's complement number. It has been shown that the inverted encoding of negabits [13] leads to direct applicability of standard full adders and other counters and compressors for any combination of posibits and negabits. Example 1 briefly describes such functionality of full adders.

Example 1 (*Full adders with mixed polarity I/O*). Fig. 1 shows four standard full adders with four different mixed-polarity (i.e., a mix of posibits and negabits) inputs. The polarity of inputs and outputs is indicated according to the conventions in Table 1. For example, the functionality of the full adder (c) is justified as follows, where the arithmetic value of an inversely encoded negabit B is $\|B\| = B-1$, that of a posibit b is $\|b\| = b$, lower- and upper-case versions of a bit-variable are assumed to have the same logical value (i.e., $b = B$), and $x+y+z = 2c+s$ describes the function of a standard full adder with input bits x, y and z, carry c and sum s:

$$\begin{aligned}\|X\| + \|Y\| + \|z\| &= X - 1 + Y - 1 + z = x + y + z - 2\\ &= 2c + s - 2 = 2(C-1) + s = 2\|C\| + \|s\|.\end{aligned}$$

Table 1
Convention for twit notation.

Twit	Dot notation	Symbolic notation
Posibit	●	x
Negabit	○	X
Unibit	▣	$\underline{X}$

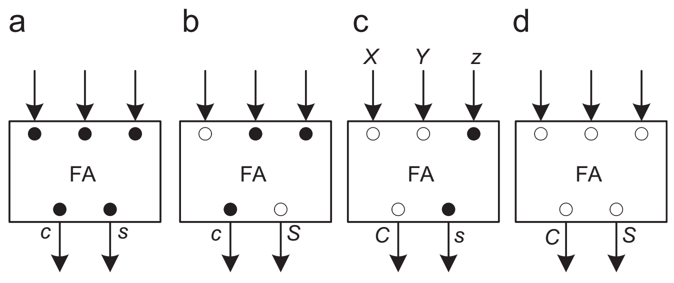


Fig. 1. Four functionality of standard full adders.

Definition 2 (*Unibit*). A unibit (i.e., unit-valued bit) $\{-1, 1\}$ is a special case of a twit with $\lambda = -1$, and $\gamma = 2$.

Definition 3 (*WTS encoding*). A WTS encoding a digit set includes an arbitrary number of twits, with arbitrary variety (i.e., different values for λ and γ), in each binary position.

Definition 4 provides an example of WTS encoding.

Definition 4 (*Decimal-SUT*). A decimal stored-unibit-transfer digit represents the interval $[-8, 9]$. The encoding is composed of a transfer part and a main part. The transfer part is a weighted-1 unibit and the main part contains a 4-bit inverted-polarity two's complement number consisting of a weighted-8 posibit, and 3 negabits weighted 4, 2, and 1.

Fig. 2 depicts a decimal-SUT digit in $[-8, 9]$. The convention used in this paper for graphic and symbolic notation of twits is shown in Table 1. Posibits, negabits, and unibits are shown by lowercase, uppercase, and underlined uppercase letters, respectively.

The encoding of decimal-SUT digits is quite similar to the original SUT-digit definition [11] with radix-16 digit set $[-9, 8]$ (i.e., ○●●● as the main part and a unibit as the transfer). The rationale for enforcing the latter is twofold:

- We have deliberately used inverse polarities in the main part in order to overcome difficulty (a), listed in Section 2. This limits the positive range of the proposed digit set to be exactly equal to [0, 9], which obviates the need for over-9 detection. Note that the maximum positive value (i.e., 9) occurs when both the posibit and the unibit are 1.
- As regards the negative range, the required redundancy for carry-free addition [12] may be provided by only two negative values (i.e., -1 and -2). However, we have included six extra negative values (i.e., $[-8, -3]$) to make the digit set minimally asymmetric. This allows us to use an adder similar to the SUT unified adder/subtractor of [11], thus surmounting difficulty (c), listed in Section 2.

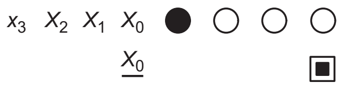


Fig. 2. Proposed encoding for redundant decimal digit.

The proposed decimal-SUT addition scheme, as described in the next section, is shown to lead to faster add/subtract operation with respect to the three previous fully redundant schemes mentioned at the end of Section 3. This is due to the intrinsic efficiency of stored transfer representation. As regards the difficulty (b), enumerated in Section 2, we show that the correction step in the proposed addition scheme is no more complex than other alternatives.

5. Decimal-SUT adder

An addition scheme for decimal-SUT numbers is described by Algorithm 4, where superscripted letters denote decimal digits. The actual realization of the algorithm is simpler than its five steps may suggest. The dot-notation and symbolic representation of the algorithm and an abstract building block of the adder, as depicted in Fig. 3 demonstrate its simplicity.

Algorithm 4 (*Decimal-SUT addition*).
Inputs: Two k-digit decimal-SUT numbers $X = x^{k-1},\ldots,x^0$ and $Y = y^{k-1},\ldots,y^0$, where $x^i = (x_3^i X_2^i X_1^i X_0^i, \underline{X}_0^i)$ and $y^i = (y_3^i Y_2^i Y_1^i Y_0^i, \underline{Y}_0^i)$, for $0 \le i \le k-1$.
Output: A k-digit decimal-SUT number $S = s^{k-1},\ldots,s^0$, where $s^i = (s_3^i S_2^i S_1^i S_0^i, \underline{S}_0^i)$ for $0 \le i \le k-1$.
Perform the following digit operations for all positions i $(0 \le i \le k-1)$ concurrently:

I. Form the 4-bit inverted-polarity two's complement sum $z^i = z_3^i Z_2^i Z_1^i Z_0^i = \underline{X}_0^i + \underline{Y}_0^i$.
II. Compute the carry-save sum $(u_4^i U_3^i U_2^i U_1^i,\ v_3^i V_2^i V_1^i V_0^i) = x^i + y^i + z^i$, using a 4-bit binary carry-save adder.
III. Compute $C_{i+1}\ w_3^i W_2^i W_1^i W_0^i = U_3^i U_2^i U_1^i 0 + v_3^i V_2^i V_1^i V_0^i$ using a 3-bit binary adder.
IV. Compute $s_3^i S_2^i S_1^i S_0^i = w_3^i W_2^i W_1^i W_0^i + 6 \times (C_{i+1} + u_4^i)$, where the parenthesized sum determines the signed-carry (in $\{-1, 0, 1\}$) into decimal position i+1.
V. Add the three equally weighted twits W_0^i, C_i and u_4^{i-1}, to compute S_0^i and $\underline{S}_0^i$.

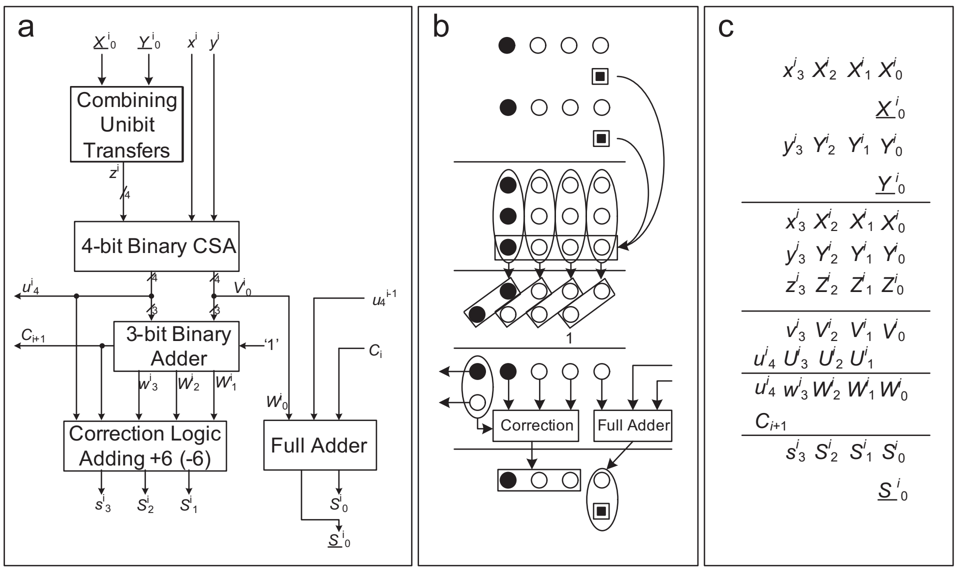

a
$\underline{X}_0^i$ $\underline{Y}_0^i$ x^i y^i
Combining Unibit Transfers
z^i
4
4-bit Binary CSA
4
4
u_4^i
V_0^i
3
3
C_{i+1}
3-bit Binary Adder
'1'
u_4^{i-1}
C_i
w_3^i W_2^i W_1^i
W_0^i
Correction Logic Adding +6 (-6)
Full Adder
s_3^i S_2^i S_1^i
S_0^i
$\underline{S}_0^i$
b
1
Correction
Full Adder
c
x_3^i X_2^i X_1^i X_0^i
$\underline{X}_0^i$
y_3^i Y_2^i Y_1^i Y_0^i
$\underline{Y}_0^i$
x_3^i X_2^i X_1^i X_0^i
y_3^i Y_2^i Y_1^i Y_0^i
z_3^i Z_2^i Z_1^i Z_0^i
v_3^i V_2^i V_1^i V_0^i
u_4^i U_3^i U_2^i U_1^i
u_4^i w_3^i W_2^i W_1^i W_0^i
C_{i+1}
s_3^i S_2^i S_1^i S_0^i
$\underline{S}_0^i$

Fig. 3. Abstract view of carry-free decimal adder: (a) circuit, (b) dot-notation, (c) symbolic notation.

The steps of Algorithm 4 may be justified as follows:

- *Step* I: The collective value of the two unibits may be encoded as an inverted-polarity 4-bit two's complement number to match the encoding of the main parts of the two operands (Fig. 3(b)). Fig. 4 depicts the required logic, which is justified by the encoding details shown in Table 2.
- *Step* II: The four standard full adders implementing this step work with any mix of posibits and inversely encoded negabits [13].
- *Step* III: V_0^i and u_4^i remain intact. $W_0^i = V_0^i$, and a 3-bit binary adder with enforced carry-in (representing a 1-valued negabit with arithmetic value 0) performs the addition.
- *Step* IV: The collective value of the equally weighted negabit-posibit pair C_{i+1}, and u_4^i lies in $\{-1, 0, 1\}$, with the actual worth of $\{-16, 0, 16\}$ with respect to decimal position *i*. The ± 6 correction is justified by the desire to consider this value as a decimal carry, with the worth of $\{-10, 0, 10\}$.
- *Step* V: The collective value of two negabits $W_0^i$and C_i and posibit u_4^{i-1} falls within $[-2,1]$. These twits may be combined by a full adder to form $\underline{S_0^i}$ and S_0^i. The three leftmost and the two rightmost columns of Table 3, which represent a truth table of a full adder, justify the latter functionality of a standard full adder.

The correction block of Fig. 3(a) may be implemented by the equation-set (1). This equation-set has been derived to meet the argument given in the justification of Step IV of Algorithm 4 above, and checked for correctness through exhaustive VHDL simulation. The delay, as shown by the corresponding gate-level

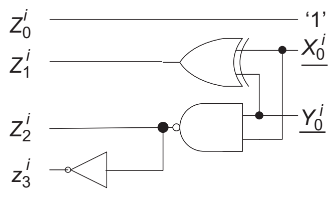


Fig. 4. Circuit for combining unibit transfers.

Table 2
Combining unibit-transfers.

$\underline{X_0^i}$	$\underline{Y_0^i}$	Sum	Z_3^i	Z_2^i	Z_1^i	Z_0^i
0	0	−2	0	1	0	1
0	1	0	0	1	1	1
1	0	0	0	1	1	1
1	1	2	1	0	0	1

Table 3
Truth table for $\underline{S_0^i}$and S_0^i.

C_i	U_4^{i-1}	W_0^i	Sum	$\underline{S_0^i}$	S_0^i
0	0	0	−2	0	0
0	0	1	−1	0	1
0	1	0	−1	0	1
0	1	1	0	1	0
1	0	0	−1	0	1
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	1	1	1

implementation (Fig. 5), is equal to that of two logic levels:

$$
\begin{aligned}
s_3^i &= w_3^i u_4^i + w_3^i C_{i+1} + w_3^i W_2^i W_1^i + u_4^i C_{i+1} W_1^i W_2^i \\
S_2^i &= (\overline{u_4^i}\,\overline{C_{i+1}}\,\overline{W_2^i} W_1^i + u_4^i C_{i+1} \overline{W_2^i}\,\overline{W_1^i}) + (\overline{u_4^i} C_{i+1} W_2^i + \overline{C_{i+1}} W_2^i \overline{W_1^i} + u_4^i W_2^i W_1^i) \\
S_1^i &= \overline{u_4^i} C_{i+1} W_1^i + \overline{u_4^i}\,\overline{C_{i+1}}\,\overline{W_1^i} + u_4^i \overline{C_{i+1}}\, W_1^i + u_4^i C_{i+1} \overline{W_1^i} = \overline{u_4^i \oplus C_{i+1} \oplus W_1^i}.
\end{aligned} \tag{1}
$$

For the purposes of delay analysis, and fair comparison, we provisionally follow the approach of the previous works (i.e., [26,19]), where the latency of simple three-input gates is assumed to be 1ΔG and that of an XOR gate is 2ΔG. Then the upper bound for overall delay of the adder of Fig. 3, based on the following delay components, amounts to 14ΔG. However, a more realistic analysis based on logical effort [27] is offered in Section 8.

(a) Combining the unibit transfers (Fig. 4): 2ΔG.
(b) Carry-save adder (only one of the XOR gates is in the critical path): 2ΔG.
(c) Three-bit adder preferably implemented by a carry acceleration technique (e.g., carry look-ahead (CLA)): 6ΔG, due to two 3-input gates for carry calculation and one XOR gate.
(d) Correction logic (Fig. 5): 4ΔG (based on three-input gates).

6. Carry-free subtraction of decimal-SUT numbers

It is often desirable to use the addition circuitry for subtraction, as is the case in the typical general-purpose two's complement processors, where the subtraction penalty is one XOR gate per bit. The general idea is to negate the subtrahend by bit-wise inversion followed by an enforced carry addition. In nonredundant decimal arithmetic, however, the subtraction is more involved and the penalty is considerable [24]. A minimal penalty solution (i.e., one XOR gate per bit) is offered in [26,19], where a symmetric redundant representation is used for decimal digits (i.e., [−7, 7] and [−9, 9], respectively). The proposed redundant decimal digit set of this work (i.e., [−8, 9]) is minimally asymmetric and also leads to minimal penalty subtraction. This claim is supported by the following lemma:

Lemma 1 (*Twit-wise inversion of decimal-SUT digits*). *Twit-wise inversion of a decimal-SUT digit in* [−8, 9] (see Definition 4) *leads to the Excess-1 negation of the original digit.*

Proof. . The arithmetic value of a posibit *x*, a negabit *X*, and a unibit *X*, given our special encoding of twits (Definitions 1 and 2), is *x*, (−1+*X*), and (−1+2*X*), respectively. The following equations clarify the lemma's claim, where ‖*D*‖ denotes the arithmetic value of a digit *D*:

$$
\begin{aligned}
\|x_3 X_2 X_1 X_0, \underline{X_0}\| &= 8x_3 + 4(-1 + X_2) + 2(-1 + X_1) - 1 + X_0 - 1 + 2\,\underline{X_0} \\
&= 8x_3 + 4X_2 + 2X_1 + X_0 + 2\,\underline{X_0} - 8 \\
\|\overline{x_3}\,\overline{X_2}\,\overline{X_1}\,\overline{X_0}, \overline{\underline{X_0}}\| &= 8(1 - x_3) - 4X_2 - 2X_1 - X_0 + 1 - 2\,\underline{X_0} \\
&= -(8x_3 + 4X_2 + 2X_1 + X_0 + 2X_0 - 8) + 1 \\
&= -\|x_3 X_2 X_1 X_0, \underline{X_0}\| + 1.
\end{aligned}
$$

□

Corollary 1 (*Negation of a decimal-SUT digit*). *To negate a decimal-SUT digit, it is sufficient to invert all its twits and store a negabit, bearing arithmetic value −1 (logical 0), along its least significant position.*

Given the above simple negation technique, we can easily adapt the decimal-SUT adder to perform subtraction. We just XOR the twits of the second operand (i.e., addend or subtrahend) with an

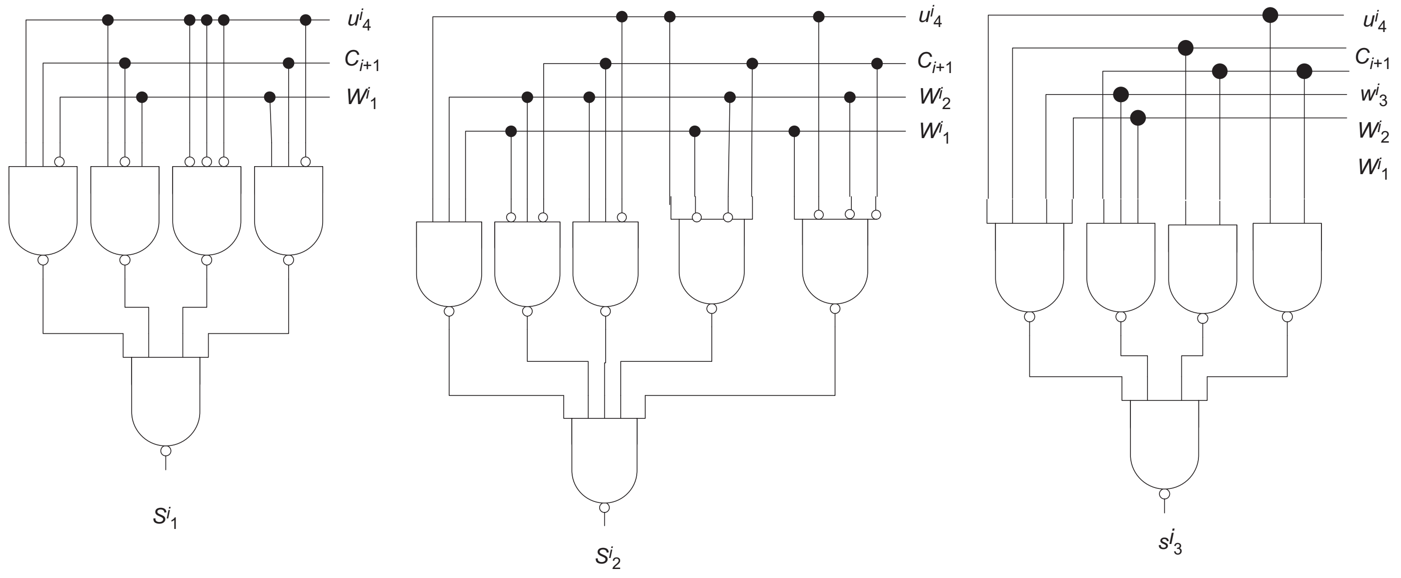


Fig. 5. Correction logic.

Table 4
Conversion from BCD to Decimal-SUT.

Value	a_3	a_2	a_1	a_0	x_3	X_2	X_1	X_0	$\underline{X}_0$
0	0	0	0	0	1	0	0	0	0
1	0	0	0	1	1	0	0	1	0
2	0	0	1	0	1	0	1	0	0
3	0	0	1	1	1	0	1	1	0
4	0	1	0	0	1	1	0	0	0
5	0	1	0	1	1	1	0	1	0
6	0	1	1	0	1	1	1	0	0
7	0	1	1	1	1	1	1	1	0
8	1	0	0	0	1	1	1	0	1
9	1	0	0	1	1	1	1	1	1

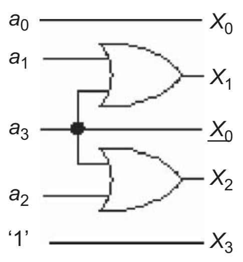


Fig. 6. BCD to decimal-SUT converter.

operation signal s (0 for addition and 1 for subtraction), and feed a negabit $\bar{S}$ (1 for addition and 0 for subtraction) instead of the enforced 1 of the 3-bit adder of Fig. 3. Therefore, the overall add/subtract latency amounts to 16ΔG.

7. Conversion from/to the conventional BCD format

Given that the decimal data are generally stored in BCD format, the BCD input operands are to be converted to decimal-SUT encoding before feeding into the proposed adder/subtractor. The decimal-SUT encoded result should in turn be converted back to BCD format.

The BCD to decimal-SUT conversion is outlined in Table 4, where $a_3a_2a_1a_0$ denotes the twits of the BCD input and ($x_3x_2x_1x_0$, X_0) represents the decimal-SUT output. Given the redundancy of decimal-SUT encoding, there may be more than one representation for each decimal-SUT digit (e.g., 01101 is another encoding for 0 that is not used in Table 4). We have taken advantage of this flexibility to design a simple conversion logic with a single gate delay (Fig. 6).

The reverse conversion, as is expected for any redundant representation, involves carry propagation across the word-width. Algorithm 5 describes the reverse-conversion steps.

Algorithm 5 (*Conversion from Decimal-SUT to BCD*).
Input: k-digit decimal-SUT number $X = x^{k-1},\ldots,x^0$, where $x^i = (x^i_3X^i_2X^i_1X^i_0, \underline{X}^i_0)$ for $0 \le i \le k-1$.
Output: k-digit BCD number $A = a^{k-1},\ldots,a^0$, where $a^i = a^i_3a^i_2a^i_1a^i_0$ for $0 \le i \le k-1$.
Set $C_0 = 1$, and perform the following digit operations for all positions i $(0 \le i \le k-1)$, where $\hat{x}^i = x^i_3X^i_2X^i_1X^i_0$:

I. Conversion of $\underline{X}^i_0 \in \{-1, 1\}$ to $u^i = U^i_3u^i_2u^i_1u^i_0 = \underline{X}^i_0\overline{\underline{X}^i_0}\,\overline{\underline{X}^i_0}\,1$.
II. Four-bit binary addition $b^i = C_{i+1}w^i = \hat{x}^i + u^i + C_i$, where $-9 \le b^i \le 9$, and $w^i = w^i_3w^i_2w^i_1w^i_0$.
III. If $C_{i+1} = 0$ (i.e., arithmetically -1) then $a^i = w^i - 6$ else $a^i = w^i$.

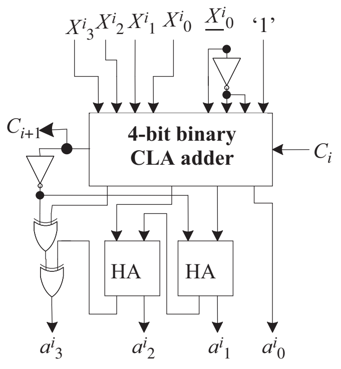
Justifications:

Step I: A unibit $\underline{X}$ can be expressed as a 2-bit two's complement number (X 1). This can be further sign extended to four bits ($X\ x\ x$ 1).

Step II: The four-bit binary adder is a cascade of four full adders of type (c) in Fig. 1. Note that C_0 is assumed as a negabit with arithmetic value 0. Therefore, the first full adder is of type (c) that generates a negabit carry-out. The same reasoning applies to the rest of full adders.

Step III: In Step II if $C_{i+1} = 0$ then $w^i = b^i + 16 \ge 7$. This leads, in Step III, to $a^i \ge 1$, i.e., the subtract-6 operation will not generate another carry.

Fig. 7 depicts a high-level implementation of the conversion Algorithm 5, where the critical-path latency for C_i to C_{i+1} is as low as 2ΔG, provided that a carry look-ahead logic is used. The overall carry-ripple delay through the least significant ($k-1$)-digits amounts to (4+2($k-2$))ΔG, and the last conversion cell shows a delay of 8ΔG. Therefore the total conversion delay is (2k+8)ΔG,



$X^i_3 X^i_2 X^i_1 X^i_0$ $\underline{X^i_0}$ '1'
C_{i+1}
4-bit binary CLA adder
C_i
HA
HA
a^i_3 a^i_2 a^i_1 a^i_0

Fig. 7. A digit slice of the decimal-SUT to BCD conversion circuit.

which may be reduced to O(log *k*) latency by using carry accelerating techniques between the conversion cells.

8. Comparison with previous works

We have encountered, in the open literature, three different designs for fully redundant carry-free decimal addition that use decimal signed-digit sets. However, we have not come across any fully redundant addition scheme based on positive decimal redundant digit sets. These have been often used for semi-redundant addition schemes (e.g., within multipliers in [16,29]). One reason could be that such digit sets introduce additional problems where both addition and subtraction are equally used (e.g., redundant digit floating point add/subtract logic [6]). Therefore we only compare the proposed scheme with the three fully redundant ones.

We have used the Logical Effort model [27] for static standard CMOS gates to evaluate area-time measures. We only wish to roughly compare the performance of the proposed adder to those of prior works, and do not aim at precise evaluation results. Therefore, we do not undertake optimizing techniques such as gate sizing, and do not consider the effect of interconnections. We rather allow gates with the drive strength of the minimum-sized inverter, and assume equal input and output loads. The latency is measured in FO4 units (i.e., the delay of an inverter with a fan-out of four inverters), and minimum-size NAND2 gate units are assumed for area evaluation.

8.1. Svoboda approach

The very early method by Svoboda, in 1969 [28], proposed a decimal signed-digit set [−6, 6], where each digit *p* in [0, 6] (*n* in [−6, 0]) is represented by the 5-bit binary encoding for 3*p* (31−3*n*). This particular encoding and the proposed carry-free addition algorithm, although interesting and innovative as the first effort in redundant-digit decimal addition, are rather inefficient in comparison with later approaches to be explained below. Moreover, the proposed BCD to redundant decimal conversion is not carry-free, and the carry-free addition scheme has four rather complex steps. Finally, Svoboda does not provide any proposal for subtraction.

8.2. Redundant BCD (RBCD)

Twenty years after Svoboda's proposal, Shirazi et al. [26] used a 4-bit two's complement encoding, called redundant binary-coded-decimal (RBCD), to represent a redundant decimal digit set [−7, 7]. The RBCD adder is designed based on the conventional carry-free addition algorithm (see Algorithm 3), where two 4-bit binary adders take care of Steps I and III, and two PLAs are responsible for decimal correction (see Step II of Algorithm 1).

The overall latency, as they have figured based on 3-input gates, amounts to 18ΔG. The RBCD adder is adapted for performing subtraction using one XOR gate per bit for two's complementing the digits of the subtrahend. Therefore, the overall latency of RBCD adder/subtractor amounts to 20ΔG. Our delay analysis of the same circuit based on logical effort shows 19.70 and 23.51 in FO4 units for addition and subtraction, respectively.

The BCD to RBCD conversion logic proposed in [26] is composed of a PLA and a 4-bit adder in sequence, leading to a slow nonredundant-to-redundant conversion process with 9ΔG delay.

The reverse conversion, like any other redundant-to-nonredundant conversion is a carry-propagating process. The reverse converter uses two PLAs and a 4-bit adder per decimal digit. The overall delay, for a *k*-RBCD-digit result, amounts to (2*k*+10)ΔG, where the carry-propagation latency of a 4-bit adder is assumed to be 2ΔG.

8.3. Decimal signed-digit (DSD) adder/subtractor

Nikmehr et al. [19] offered a decimal signed-digit adder/subtractor using the maximally redundant decimal digit set [−9, 9], represented by two equally weighted BCD digits with opposite polarities. The overall delay of the DSD adder/subtractor is evaluated, in [19], to be equal to that of [26]. However, the addition and subtraction latencies, for the same circuits, are computed as 24.62 and 28.43 in FO4 units, respectively.

Although negation is simply performed by bit inversion, the subtraction penalty is the same in terms of latency (i.e., one XOR gate per bit). However, in terms of gate count, twice as many XOR gates are used compared to [26]. Moreover the main addition operation speculatively produces six sum values that require additional branching, thus greatly increasing the area consumption and latency. The BCD to DSD conversion is a zero-time operation, an impressive improvement over the 9ΔG latency of the similar conversion in [26]. The reverse conversion is possible by subtracting the negative component from the positive one by a standard BCD subtractor [24]. Therefore, the latency of the reverse conversion, for a *k*-DSD result, amounts to (4*k*+4)ΔG.

8.4. Decimal-SUT adder/subtractor

The latency of the proposed decimal-SUT adder/subtracter, as explained in Section 5, is 16ΔG. The main improvement, regarding previous approaches, comes from the stored transfer addition scheme of Algorithm 4.

The simple BCD to decimal-SUT conversion logic (Fig. 6) uses one gate in its critical path, leading to conversion latency of only 1ΔG. The reverse conversion of a *k*-digit decimal-SUT number to its BCD equivalent (logic of Fig. 7) shows a latency of (2*k*+8)ΔG.

To summarize the above discussion on the latency of redundant-digit decimal adder/subtractor, we provide Table 5,

Table 5
Comparison between the decimal-SUT scheme and the best previous results.

Addition scheme	Adder				Adder/subtractor			
	T_{FO4}	Ratio	$Area_{NAND2}$	Ratio	T_{FO4}	Ratio	$Area_{NAND2}$	Ratio
Decimal SUT	15.32	1	125	1	16.90	1	131	1
RBCD [26]	19.70	1.28	190	1.52	23.51	1.39	200	1.52
DSD [19]	24.62	1.60	642	5.13	28.43	1.68	645	4.92

where this work is compared with that of Shirazi et al. [26] and Nikmehr et al. [19]. The latency and area improvements of the decimal-SUT unified adder/subtractor with respect to that of Shirazi et al. [26] ([19]) are 28% (34%) and 40% (79%), respectively. For a fair comparison of the three methods, we have replaced the PLAs used in [26] with equivalent combinational logic.

9. Conclusions

We observed that in digital binary arithmetic there are multi-operand addition, multiplication, and division schemes based on semi-redundant adders and subtractors. Nevertheless, fully redundant adder/subtractors are also used in multiplication and arithmetic function evaluation. In decimal arithmetic, however, we have encountered three fully redundant addition/subtraction schemes [26,28,19] that have not found tangible applications due to low efficiency in comparison with semi-redundant counter-parts. However, since fully redundant adders lead to VLSI-friendly recursive structure of a partial product tree, we were encouraged to explore faster carry-free addition/subtraction schemes. Nevertheless, there are specific applications such as decimal CORDIC arithmetic that intrinsically require fully redundant addition/subtraction that could benefit from the results of this work and future similar research.

We introduced the decimal-SUT encoding of decimal digits using the digit set $[-8, 9]$, where the transfer digit, generated through the implementation of carry-free addition algorithm, is stored with the next higher-weighted digit. Based on the logical effort analysis, the fully redundant decimal-SUT adder/subtractor is considerably faster with much less area with respect to the previous works. The BCD to redundant decimal conversion of [19] and our scheme show considerable improvement, in terms of latency and area, over that of [26]. However, the cost of reverse conversion, in all the three works, is almost the same.

The prospect for further research includes exploration of possible more efficient fully redundant decimal addition/subtraction schemes and encodings of decimal digits and their application in the design and implementation of decimal function evaluation hardware.

References

[1] A. Avizienis, Signed-digit number representations for fast parallel arithmetic, IRE Trans. Electron. Comput. 10 (1961) 389–400.
[2] M.F. Cowlishaw, Decimal floating-point: algorism for computers, in: Proceedings of the 16th IEEE Symposium on Computer Arithmetic, June 2003, pp. 104–111.
[3] L. Dadda, Multi operand parallel decimal adder: a mixed binary and BCD approach, IEEE Trans. Comput. 56 (10) (2007) 1320–1328.
[4] M.D. Ercegovac, T. Lang, Digital Arithmetic, Morgan Kaufmann Publishers, Los Altos, CA, 2004.
[5] M.A. Erle, E.M. Schwartz, M.J. Schulte, Decimal multiplication with efficient partial product generation, in: 17th IEEE Symposium on Computer Arithmetic (ARITH-17), 2005, pp. 21–28.
[6] H.A.H. Fahmy, M.J. Flynn, The case for a redundant format in floating point arithmetic, in: Proceedings of the 16th IEEE Symposium on Computer Arithmetic, Santiago de Compostela, Spain, June 2003.
[7] A.F. Gonzalez, P. Mazumder, Redundant arithmetic, algorithms, implementations, Integration VLSI J. 30 (2000) 13–53.
[8] Institute of Electrical and Electronics Engineers, Draft IEEE Standard for Floating-Point Arithmetic, 2007.
[9] G. Jaberipur, B. Parhami, M. Ghodsi, A class of stored-transfer representations for redundant number systems, in: Proceedings of 35th Asilomar Conference on Signals Systems and Computers, 2001, pp. 1304–1308.
[10] G. Jaberipur, B. Parhami, M. Ghodsi, Weighted bit-set encodings for redundant digit sets: theory and applications, in: Proceedings of 36th Asilomar Conference on Signals, Systems and Computers, 2002, pp. 1629–1633.
[11] G. Jaberipur, B. Parhami, M. Ghodsi, Weighted two-valued digit-set encodings: unifying efficient hardware representation schemes for redundant number systems, IEEE Trans. Circuits Syst. I 52 (7) (2005) 1348–1357.
[12] G. Jaberipur, B. Parhami, Stored-transfer representations with weighted digit-set encodings for ultrahigh-speed arithmetic, IET Circuits Devices Syst. 1 (1) (2007) 102–110.
[13] G. Jaberipur, B. Parhami, Constant-time addition with hybrid-redundant numbers: theory and implementations, Integration VLSI J. 41 (1) (2008) 49–64.
[14] R.D. Kenney, M.J. Schulte, M.A. Erle, A high-frequency decimal multiplier, in: IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD), 2004, pp. 26–29
[15] R.D. Kenney, M.J. Schulte, High-speed multioperand decimal adders, IEEE Trans. Comput. 54 (8) (2005) 953–963.
[16] T. Lang, A. Nannarelli, A radix-10 combinational multiplier, in: Proceedings of Asilomar Conference on Signals, Systems, and Computers, 2006, pp. 313–317.
[17] T. Lang, A. Nannarelli, A radix-10 digit-recurrence division unit: algorithm and architecture, IEEE Trans. Comput. 56 (6) (2007) 727–739.
[18] G. Metze, J.E. Robertson, Elimination of Carry propagation in digital computers, in: Proceedings of International Conference on Information Processing, Paris, 1959, pp. 389–396.
[19] H. Nikmehr, B.J. Phillips, C.C. Lim, A decimal carry-free adder, in: Proceedings of SPIE Conference on Smart Materials, Nano-, Micro-Smart Systems, 2004, pp. 786–797.
[20] H. Nikmehr, B. Phillips, C.C. Lim, Fast decimal floating-point division, IEEE Trans. Very Large Scale Integration (VLSI) Syst. 14 (9) (2006) 951–961.
[21] B. Parhami, Generalized signed-digit number systems: a unifying framework for redundant number representations, IEEE Trans. Comput. 39 (1) (1990) 89–98.
[23] R.K. Richards, Arithmetic Operations in Digital Computers, Van Nostrand Comp., Inc., 1955.
[24] M. Schmookler, A. Weinberger, High speed decimal addition, IEEE Trans. Comput. C-20 (8) (1971) 862–866.
[25] S. Shankland, IBM's POWER6 gets help with math, multimedia, ZDNet News (2006).
[26] B. Shirazi, D.Y. Yun, C.N. Zhang, RBCD: redundant binary coded decimal adder, in: IEE Proceedings on Computer & Digital Techniques (CDT), vol. 36, no. 2, 1989.
[27] I.E. Sutherland, R.F. Sproull, D. Harris, Logical Effort: Designing Fast CMOS Circuits, Morgan Kaufmann, Los Altos, CA, ISBN 1558605576, 1999.
[28] A. Svoboda, Decimal adder with signed digit arithmetic, IEEE Trans. Comput. C-18 (3) (1969) 212–215.
[29] A. Vazquez, E. Antelo, P. Montuschi, A new family of high-performance parallel decimal multipliers, in: Proceedings of the 18th IEEE Symposium on Computer Arithmetic, 2007, pp. 195–204.
[30] L. Wang, M.J. Schulte, A decimal floating-point divider using Newton–Raphson iteration, J. VLSI Signal Process. Syst. 14 (1) (2007) 3–18.
[31] C.K. Yuen, A new representation for decimal numbers, IEEE Trans. Comput. C-26 (12) (1977) 1286–1288.

Amir Kaivani received his B.S. in Computer Engineering from Islamic Azad University of Mashhad in 2005, and M.S. in Computer Engineering from Shahid Beheshti University (SBU) in 2007. He is currently a Ph.D. candidate in Electrical Engineering at SBU. His research interests include computer arithmetic, quantum computing and reversible circuit design.

Ghassem Jaberipur, associate professor of Computer Engineering in the Department of Electrical and Computer Engineering of Shahid Beheshti University (Tehran, Iran), received his B.S in Electrical Engineering and Ph.D. in Computer Engineering from Sharif University of Technology in 1974 and 2004, respectively, M.S in Engineering from UCLA in 1976, and M.S in Computer Science from University of Wisconsin in Madison in 1979. Currently, his main research interest is in Computer Arithmetic.