

## Fully Redundant Decimal Arithmetic

Saeid Gorgin and Ghassem Jaberipur

*Dept. of Electrical & Computer Engr., Shahid Beheshti Univ. and  
School of Computer Science, institute for research in fundamental sciences (IPM), Tehran, Iran  
Gorgin@sbu.ac.ir, Jaberipur@sbu.ac.ir*

### Abstract

*Hardware implementation of all the basic radix-10 arithmetic operations is evolving as a new trend in the design and implementation of general purpose digital processors. Redundant representation of partial products and remainders is common in the multiplication and division hardware algorithms, respectively. Carry-free implementation of the more frequent add/subtract operations, with the byproduct of enhancing the speed of multiplication and division, is possible with redundant number representation. However, conversion of redundant results to conventional representations entails slow carry propagation that can be avoided if the results are kept in redundant format for later use as operands of other arithmetic operations. Given that redundant decimal representations, contrary to redundant binary, do not necessarily require extra storage, we are motivated to develop a framework for fully redundant decimal arithmetic, where all operands and results belong to the same redundant decimal number system and can be stored and later used as operands of further decimal operations. In this paper, we present a new faster decimal signed digit add/sub unit and show how it can be efficiently used in the design of decimal multipliers and dividers, where all operands and results are represented with the same redundant digit set  $[-7, 7]$ .*

### 1. Introduction

Decimal computer arithmetic and the supporting hardware units are once again in the forefront of commercial, financial, scientific, and internet-based applications [1]. The current trend is mirrored, in the industry, by commercialization of digital processors with embedded decimal arithmetic units (e.g., IBM z900 eServer [2], power6 [3] and z10 [4]), and in the literature, via the state of the art parallel decimal multipliers (e.g., [5] and [6]), dividers (e.g., [7], [8] and [9]), function evaluation and CORDIC [10] hardware.

In both decimal and binary arithmetic, partial products in multipliers and partial remainders in dividers are often represented via a redundant number system (e.g., Binary signed digit [11], decimal carry-save [5], double-decimal [6], and minimally redundant decimal [9]). Such use of redundant digit sets, where the number of digits is sufficiently more than the radix, allows for carry-free addition and subtraction as the basic operations that build-up the product and remainder, respectively. In the aforementioned works on decimal multipliers and dividers, inputs and outputs are nonredundant decimal numbers. However, a redundant representation is used for the intermediate partial products or remainders. The intermediate additions and subtractions are semi-redundant operations in that only one of the operands as well as the result is redundant. In contrast there are fully-redundant add/subtract schemes, where both operands and certainly the result are represented via redundant decimal digit sets (e.g., [12], [13] and [14]). Fully redundant decimal addition is also used within a sequential decimal multiplier [15], where partial products are represented in Svoboda's decimal signed digit encoding [12]. However, we have not encountered any fully redundant radix-10 multiplier or divider in the literature.

In a comprehensive study on the frequency of arithmetic operations of a general computation [16], it has been shown that add/subtract operations occur more frequently than multiplication and division. Therefore, carry-free addition/subtraction can have a great impact on the overall execution time. However, there are usually two problems:

- **Problem 1 (Storage of redundant results):**

The redundant result of an addition/subtraction is not necessarily used immediately as the operand of a subsequent operation. Therefore, it should be appropriately stored for later use. But, redundant results often require wider storage words or registers due to extra redundancy bits within a digit.

This may not be required when the radix of the number system is not a power of two. For example, in radix-10 arithmetic, at least four bits are required for representation of a 10-valued nonredundant decimal digit. Therefore, there are possibly six extra 4-bit codes available to be assigned to extra digits of a redundant decimal digit set. For instance, the digit set  $[-7, 7]$  has been used in the design of a fully redundant decimal adder [13] and for representation of quotient digits in the design of a nonredundant radix-10 divider [8], where each digit is represented as a 4-bit two's complement number. The overloaded decimal digit set  $[0, 15]$  used in [17] represents another example of a redundant decimal encoding with no extra redundancy bit. There are however, redundant decimal digit sets that use more than four bits per digit (e.g., 6 bits in [12] and 8 bits in [14]).

- **Problem 2 (Intermixed operations):**

Other operations such as multiplication and division may be intermixed with additions and subtractions. Therefore, use of conventional multipliers and dividers that require nonredundant operands enforces the conversion of redundant results, of add/subtract operations, to conventional nonredundant format. The conversion may require word wide carry propagation that may jeopardize the speed gained via carry-free add/subtract operations. This problem can be avoided if we keep the redundant results intact when they are needed as operands of other operations such as multiplication or division; hence the necessity to design fully redundant multipliers and dividers.

In this paper, a preliminary discussion on decimal addition is offered in Section 2. We propose fully redundant add and subtract schemes for a redundant decimal number system with digits in  $[-7, 7]$ , in Sections 3 and 4, respectively. To further strengthen the framework for fully redundant decimal arithmetic, without trying to be comprehensive due to space limit, we show in Sections 5 and 6 that how the proposed carry-free decimal adder and subtractor can serve as building blocks for fully redundant decimal multipliers and dividers, respectively. Section 7 is dedicated to comparisons with the previous fully redundant adders, where five different adders are synthesized based on TSMC 0.13  $\mu\text{m}$  standard CMOS technology. Also, the proposed multiplier and divider are compared with the best previous nonredundant designs. Finally, Section 8 contains our concluding remarks.

## 2. Preliminaries

### 2.1. Nonredundant decimal addition

Straightforward implementation of decimal addition/subtraction on binary digital computers is based on decimal full adders (DFA) that compute the Eqn. set 1, where  $p = x + y + c_{in}$ ,  $s$ ,  $x$  and  $y$  are decimal digits,  $c_{in}$  and  $c_{out}$  are decimal carries, and  $|A|_m$  stands for  $A$  modulo  $m$ .

$$c_{out} = \left\lfloor \frac{p}{10} \right\rfloor, s = |p|_{10} \quad (1)$$

Design of the DFA, based on a standard 4-bit binary adder, entails the tasks of over-9 detection and +6-correction. Eqn. set 2 describes the operation of a 4-bit adder, where  $w_4$  is the hexadecimal carry-out and  $w$  is the 4-bit sum.

$$w_4 = \left\lfloor \frac{p}{16} \right\rfloor, \quad w = |p|_{16} \quad (2)$$

Eqn. set 3 illustrates the aforementioned tasks that arise in the straightforward computation of  $c_{out}$  and  $s$  from  $p$  and  $w$ , respectively.

$$c_{out} = \begin{cases} 0, & \text{if } p \leq 9 \\ 1, & \text{if } p \geq 10 \end{cases}, s = \begin{cases} w, & \text{if } p \leq 9 \\ |w + 6|_{16}, & \text{if } p \geq 10 \end{cases} \quad (3)$$

Several decimal addition algorithms and their hardware realizations have been offered in the literature (e.g., the classical work in [18] and the recent one in [19]). Given that in 2's complement arithmetic the addition circuitry is used for subtraction, where the overhead is only one XOR gate per bit, the obvious challenge is to also unify, with minimal overhead, the circuitry for decimal addition and subtraction.

### 2.2. Redundant decimal addition

Let  $x_i$ ,  $y_i$ , and  $s_i$  all in  $[-\alpha, \beta]$  (e.g.,  $\alpha = \beta = 7$ ) denote the digits of the operands and result in position  $i$  (i.e., weighted  $10^i$ ), such that  $s_i = p_i + t_i - 10t_{i+1}$ , where  $[-\alpha, \beta]$ , ruled by Eqn. set 4, is the redundant decimal digit set, the position sum  $p_i = x_i + y_i$ , and  $t_{i+1}$  (similarly  $t_i$ ) is computed as in Eqn. 5, with  $\delta = \begin{cases} 0, & \text{if } \alpha > 0 \\ 1, & \text{if } \alpha = 0 \end{cases}$

$$11 \leq \alpha + \beta \leq 15, \quad \alpha, \beta \geq 0, \quad \alpha, \beta \neq 1 \quad (4)$$

$$t_{i+1} = \delta + \begin{cases} -1, & \text{if } p_i \leq -\alpha \\ 0, & \text{if } \alpha < p_i < \beta \\ 1, & \text{if } p_i \geq \beta \end{cases} \quad (5)$$

The restrictions in (4) guarantee that  $s_i \in [-\alpha, \beta]$  (see [20] for a general proof) and only four bits are enough for encoding the digit set  $[-\alpha, \beta]$ .

The straightforward implementation of Eqn. 5, as more or less followed in [13] for the balanced digit sets (i.e.,  $\alpha = \beta$ ), entails the following four steps:

- I. Compute  $p_i = x_i + y_i$ .
- II. Extract transfer  $t_{i+1}$  via comparison of  $p_i$  with  $\alpha$ .
- III. Compute the interim sum digit  $w_i = p_i - 10t_{i+1}$ .
- IV. Form the final sum digit  $s_i = w_i + t_i$

All the above steps normally experience worst case carry propagation across the 4-bit digits of the operands. More efficient implementation, however, may be envisaged by noting that the interim sum digit  $w_i$  and transfer digit  $t_{i+1}$  can be expressed and implemented in hardware directly as functions of  $x_i$  and  $y_i$ . But, this calls for the uneasy task of designing 8-input combinational logic, where the hardware cost may not be appreciated. We propose a compromise design with less complex combinational logic and shorter carry propagation chains.

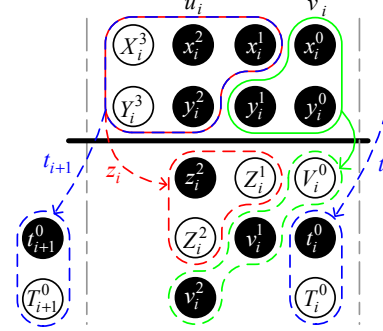
### 3. The improved decimal SD addition

We assume a redundant decimal number system with decimal signed digit set  $[-7, 7]$  and call the digit set as decimal septa signed digit (DSSD) set, to reflect the boundary values  $\alpha = \beta = 7$ . Following [13], we represent each DSSD as a 4-bit two's complement number. Step I of the decimal addition scheme, outlined in Section 2, can be implemented at no cost by representing  $p_i$  as a two's complement carry-save (TCCS) number [21]. This is illustrated in Fig. 1, where superscripts (subscripts) indicate bit (digit) weighted-2 (-10) positions. Also, white (black) circles with uppercase (lowercase) variables represent negabits (posibits).



**Figure 1.** The TCCS position sum  $p_i$ .

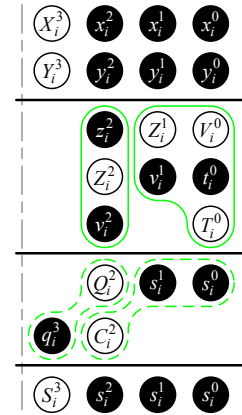
Let  $p_i = \|u_i\| + \|v_i\|$ , where  $\|b\|$  represents the arithmetic value of the bit collection  $b$ ,  $u_i$  is composed of the three most significant bits of one operand and two of the other and  $v_i$  represents the three remaining bits, as illustrated in Fig. 2. Then,  $\|v_i\| \in [0, 4]$  and  $\|u_i\| \in \{-16, -14, \dots, -2, 0, 2, \dots, 8, 10\}$ . We further decompose  $\|u_i\|$ , to the decimal transfer  $\|t_{i+1}\| \in \{-1, 0, 1\}$  and the residue  $\|z_i\| \in \{-6, -4, -2, 0, 2\}$  associated with the bit collection  $z_i$ . Therefore, the range of the interim sum  $w_i = \|z_i\| + \|v_i\|$  is  $[-6, 6] = [-6, 2] + [0, 4]$ , which leads to  $s_i \in [-7, 7]$ .



**Figure 2.** Decomposition of  $u_i$  to transfer  $t_{i+1}$  and  $z_i$

Note that in the alternative balanced decomposition, with the aim of extracting  $t_{i+1}$  from only four bits (i.e.,  $X_i^3, Y_i^3, x_i^2$  and  $y_i^2$ ), a positive residue is inevitable. On the other hand, the arithmetic value of the remaining bits falls within  $[0, 6]$ . Therefore there will be cases, where no room is left for the coming transfer  $t_i$ .

Fig. 3 illustrates the addition process that is composed of the following overlapping steps  $i$  to  $v$ , where Eqn. sets 6 to 10 are derived, via straightforward truth tables.



**Figure 3.** The complete addition process.

- i. Decomposition of  $u_i$  ruled by  $\|u_i\| = 10 \|t_{i+1}\| + \|z_i\|$  (see Fig. 2), where the constituent bits of  $t_{i+1}$  and  $z_i$  are derived as in Eqn. set 6.

$$\begin{aligned}
 t_{i+1}^0 &= \overline{X_i^3} \vee Y_i^3, \\
 T_{i+1}^0 &= X_i^3 Y_i^3 (x_i^2 \vee x_i^1 y_i^2) \vee x_i^2 \vee y_i^2 x_i^1 (X_i^3 \vee Y_i^3), \\
 z_i^2 &= X_i^3 Y_i^3 (x_i^2 x_i^1 \vee y_i^2) \vee \overline{X_i^3} Y_i^3 x_i^1 y_i^2 \vee \\
 &\quad (X_i^3 \vee Y_i^3) \oplus x_i^1 y_i^2 \vee \overline{X_i^3} x_i^2 y_i^2, \\
 z_i^1 &= \overline{X_i^3} Y_i^3 (x_i^2 \vee y_i^2 x_i^1 \vee x_i^1 y_i^2) \vee X_i^3 x_i^2 (x_i^1 \oplus Y_i^3) \vee \\
 &\quad y_i^2 (\overline{X_i^3} \vee Y_i^3 x_i^2 \vee x_i^2 x_i^1 Y_i^3), \\
 z_i^0 &= (x_i^2 \vee y_i^2 \vee x_i^1) (X_i^3 \oplus Y_i^3) \vee \overline{X_i^3} (x_i^2 \vee y_i^2 x_i^1 \vee \\
 &\quad x_i^1 Y_i^3 y_i^2) \vee Y_i^3 (x_i^2 x_i^1 y_i^2 \vee X_i^3 x_i^1). \tag{6}
 \end{aligned}$$

- ii. Transformation of  $v_i$  to  $v_i^2 v_i^1 V_i^0$  (in parallel with Step i), described by Eqn. set 7.

$$V_i^0 = x_i^0 \oplus y_i^0, v_i^1 = y_i^1 \oplus (x_i^0 \vee y_i^0), v_i^2 = y_i^1 (x_i^0 \vee y_i^0) \quad (7)$$

- iii. 2-bit addition leading to carry bit  $C_i^2$  and sum bits  $s_i^1$  and  $s_i^0$  (Eqn. set 8).

$$\begin{aligned} s_i^0 &= (\overline{V_i^0} \oplus t_i^0) \oplus \overline{T_i^0}, \\ C_i^1 &= \overline{V_i^0} \overline{T_i^0} \vee (\overline{V_i^0} \vee \overline{T_i^0}) t_i^0 = \text{carry}(\overline{V_i^0}, \overline{T_i^0}, t_i^0), \\ s_i^1 &= (v_i^1 \oplus \overline{Z_i^1}) \oplus \overline{C_i^1}, \\ C_i^2 &= (\overline{C_i^1} \vee \overline{Z_i^1}) v_i^1 \vee \overline{Z_i^1} \overline{C_i^1} = \text{carry}(\overline{Z_i^1}, \overline{C_i^1}, v_i^1) \end{aligned} \quad (8)$$

- iv. 1-bit addition in position 2 (Eqn. set 9) leading to  $q_i^3 Q_i^2$  (in parallel with iii).

$$\begin{aligned} Q_i^2 &= (\overline{Z_i^2} \oplus z_i^2) \oplus v_i^2, \\ q_i^3 &= (\overline{Z_i^2} \vee z_i^2) v_i^2 \vee \overline{Z_i^2} z_i^2 = \text{carry}(v_i^2, z_i^2, \overline{Z_i^2}) \end{aligned} \quad (9)$$

- v. Final 2-bit addition leading to sum bits  $S_i^3$  and  $s_i^2$ , as described by Eqn. set 10.

$$\begin{aligned} s_i^2 &= \overline{C_i^2} \oplus \overline{Q_i^2}, \\ C_i^3 &= \overline{C_i^2} \overline{Q_i^2} = \text{carry}(\overline{C_i^2}, \overline{Q_i^2}), S_i^3 = \overline{q_i^3 \vee C_i^3} \end{aligned} \quad (10)$$

Note that the function *carry* in Eqn. sets 8 to 10 is the carry function associated to standard full adders and half adders with possibly inverted inputs and outputs [22]. Moreover, since  $S_i^3$  is a negabit and no carry is to be generated out of position 3, the sum logic for  $S_i^3$  is simplified to a NOR gate. This is further illustrated in Fig. 4-a, where the critical delay path goes through nine logic levels, indeed a considerable improvement over the previous 18 logic level design in [13]. A more reliable Logical Effort [23] delay analysis leads to 13.56 FO4 for the latter and 7.80 FO4 for our adder. Also, in Section 7, we report the results of synthesizing both circuits by TSMC 0.13  $\mu\text{m}$  technology using Synopsis Design Compiler.

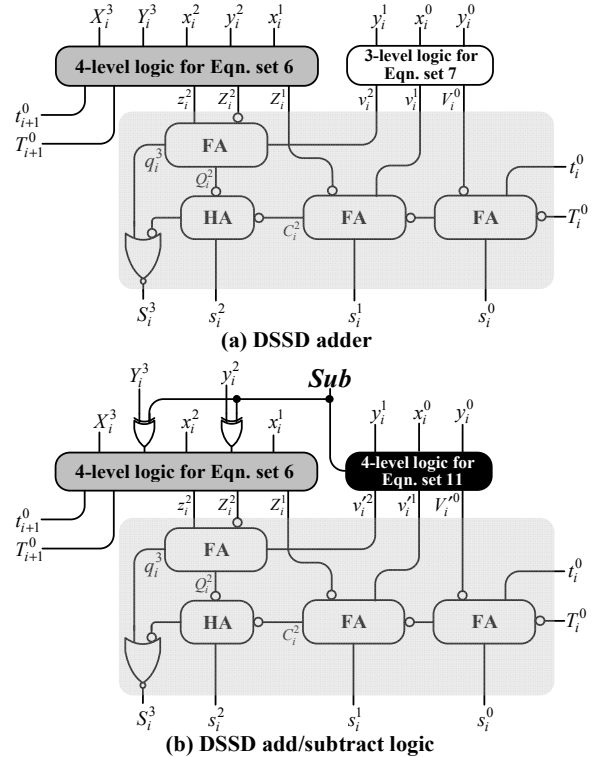
#### 4. DSSD subtractor

Since each DSSD is encoded as a 4-bit two's complement number, we simply invert all the bits of the subtrahend and perform an enforced carry addition per DSSD. The circuitry is the same as the adder of the previous section except for the logic that derives  $v_i^2$ ,  $v_i^1$  and  $V_i^0$  (see Eqn. set 7', below) corresponding to  $v_i^2$ ,  $v_i^1$  and  $V_i^0$  (see Eqn. set 7, above) and of course two inverters for the two most significant bits of the subtrahend.

$$\begin{aligned} V_i^0 &= x_i^0 \oplus y_i^0, \\ v_i^1 &= y_i^1 \oplus \overline{x_i^0 \vee y_i^0}, v_i^2 = \overline{y_i^1} \vee \overline{x_i^0 \vee y_i^0} \end{aligned} \quad (7')$$

The gate level analysis of the subtractor just described shows that the latency is equal to that of nine logic levels with 3-input gates, which is no more than that of the adder described in the previous section. Moreover, the FO4 delay is 7.80 (i.e., the same as that of the adder of Fig. 4-a). The adder (subtractor) by itself can be used as the building block for partial product reduction (partial remainder computation) in the DSSD multiplier (divider) to be briefly explained in Section 5 (6). However, in a general purpose decimal arithmetic hardware unit, it may be desired to perform subtraction using the adder circuitry. To realize such a unified add/sub unit we can use a *sub* signal to invert all the bits of the subtrahend via XOR gates, and also to select between the outputs of the circuits implementing the Eqn. sets 7 and 7'. It turns out that only the XOR gates lie within the critical delay path, thus adding two logic levels to the overall latency, which amounts to eleven logic levels and 9.5 FO4. Nevertheless, Eqn. sets 7 and 7' and their selector can be replaced by Eqn. set 11, to save hardware, as depicted in Fig. 4-b. Note that the boxes with same degree of shading in Figs. 4-a and 4-b are logically identical.

$$\begin{aligned} V_i^0 &= x_i^0 \oplus y_i^0, v_i^1 = y_i^0 (\text{sub} \oplus y_i^1) \vee \overline{y_i^0} (y_i^1 \oplus x_i^0), \\ v_i^2 &= \text{sub} y_i^1 \vee \overline{\text{sub}} y_i^1 y_i^0 \vee y_i^1 x_i^0 \overline{y_i^0} \end{aligned} \quad (11)$$



**Figure 4.** The  $i^{\text{th}}$  digit slice of the DSSD adder-only and add/sub unit.

## 5. Fully redundant DSSD multiplier

Multiplication is normally a three phase process; namely partial product generation (PPG), partial product reduction (PPR) and final product computation. The intermediate results within the PPR phase are often represented in a redundant format (e.g., binary carry-save [24], binary signed digit [11], or decimal carry-save [5]) that enables carry-free addition. The conventional multiplication techniques assume nonredundant operands and result. Therefore, the final redundant partial product needs to be converted to the nonredundant representation that is delineated for the final product. It is well known that such conversion is essentially a slow carry propagating operation. However, in fully redundant multipliers, one may use the same redundant format for input operands, partial products and the final product. Consequently, the final product is readily available as the output of the PPR phase.

In this section, we describe the PPG phase of a redundant digit decimal multiplier with DSSD operands. Although the explanations are somewhat lengthy, the hardware realization is simple. The second phase would simply make multiple use of the DSSD adder of Section 3, and directly produces the final product with DSSD digits; hence obviating the need for final carry-propagating phase.

### 5.1. PPG for DSSD operands

Decimal partial products are commonly expressed as unevaluated sum of two decimal numbers. For example, in conventional decimal multipliers (e.g., [25]), the required multiples are expressed as unevaluated sum of two *easy multiples*; that is  $(0, 1, 2, 4, 5) \times \text{multiplicand } X$ . The easy multiples are precomputed, carry-freely, as single decimal numbers. As another example, [5 and 6] precompute  $(\pm 1, \pm 2, 5, 10) \times X$ , and more recently, [26] precomputes  $(0, 1, 2, 5, 8, 9) \times X$ . To reduce the selection cost, it is naturally desirable to restrict the number of precomputed carry-free multiples to as few as possible.

For the DSSD multiplier the set of precomputed multiples  $\Pi = \{\pm X, \pm 3X, \pm 4X\}$  is a minimum set, where the rest of the required multiples can be expressed as two DSSD numbers as follows:

$$\begin{aligned} \pm 2X &= \pm X \pm X, \pm 5X = \pm 4X \pm X, \\ \pm 6X &= \pm 3X \pm 3X, \pm 7X = \pm 4X \pm 3X \end{aligned} \quad (12)$$

We provide a carry-free logic to precompute the  $\Pi$  multiples as two restricted DSSD numbers and add them by a simple 4-level logic that produces single DSSD multiples. It can be shown that the same procedure, if applied to other minimum set of multiples (e.g.,  $\pm X, \pm 2X, \pm 5X$ ), is not as efficient.

Fig. 5 depicts the required architecture, where  $h_i^m = \lfloor \frac{m \times \mu_i}{10} \rfloor$  and  $l_i^m = |m \times \mu_i|_{10}$  are the two digits of the precomputed  $m \times |x_i|$  ( $m \in \{\pm 1, \pm 3, \pm 4\}$ ) and  $\mu_i = |x_i|$ . The 3-bits of  $\mu_i$  can be computed via a simple 2-level logic. The multiplexers lead the appropriate high or low digit  $h_i^m$  or  $l_i^m$  to the final selector (expanded in Fig. 6) that is controlled by a decoder ruled by Eqn. set 13, where  $v_j = |y_j|$ ,  $\sigma_i = \text{sign}(x_i)$ ,  $\zeta_j = \text{sign}(y_j)$ ,  $h_{i,j} + h'_{i,j} = \lfloor \frac{x_i \times y_j}{10} \rfloor$ , and  $l_{i,j} + l'_{i,j} = |x_i \times y_j|_{10}$ .

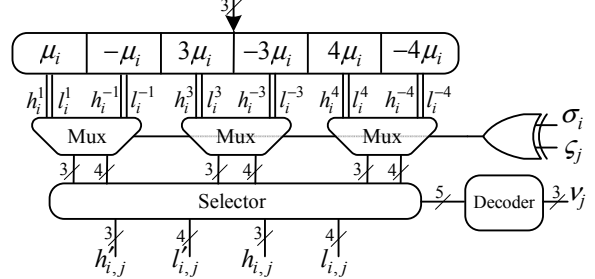


Figure 5. A digit-slice for  $x_i \times y_j$  of PPG network.

$$\begin{aligned} s^{\pm 3} &= v_i^2 v_i^0 \vee v_i^2 v_i^1, s^{\pm 1} = \overline{v_i^1} v_i^0 \vee \overline{v_i^2} v_i^1 \overline{v_i^0}, \\ s'^{\pm 4} &= v_i^2 v_i^1 \vee v_i^2 v_i^0, s'^{\pm 3} = v_i^2 v_i^1 \overline{v_i^0}, s'^{\pm 1} = \overline{v_i^2} v_i^1 \overline{v_i^0} \end{aligned} \quad (13)$$

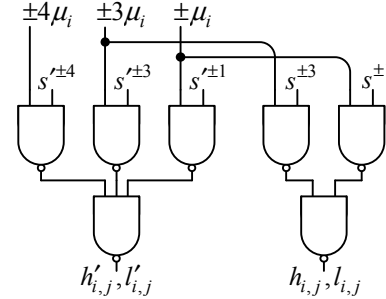


Figure 6. The logic for the selector of Figure 5.

Table I shows the complete  $h_i^m$  and  $l_i^m$  values, where  $h_i^m \in [-3, 3]$  and  $l_i^m \in [-5, 5]$ . Table II contains logical equations for the three bits of  $h_i^m$  and the four bits of  $l_i^m$  in terms of the bits of  $\mu_i$ . These equations have been easily derived via straightforward 3-bit truth tables.

To see the carry-free nature of the computation of single DSSD multiples, let  $X = x_{k-1} \dots x_0$  be the  $k$ -digit multiplicand and  $P_j + P'_j = X \times y_j$  denote the  $k+1$ -digit  $j^{\text{th}}$  partial product, where  $P_j = p_{k,j} \dots p_{0,j}$ ,  $P'_j = p'_{k,j} \dots p'_{0,j}$ , with  $p_{i,j}$  and  $p'_{i,j}$ , defined as:

$$\begin{aligned} p_{i,j} &= h_{i-1,j} + l_{i,j}, p'_{i,j} = h'_{i-1,j} + l'_{i,j} \quad (0 \leq i \leq k-1), \\ p_{k,j} &= h_{k-1,j}, p'_{k,j} = h'_{k-1,j} \end{aligned} \quad (14)$$

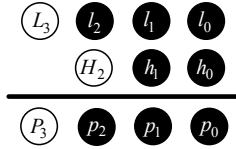
It can be easily explored from Table I that  $h_{i-1}^m + l_i^m \in [-7, 7]$  leading to  $p_{i,j}, p'_{i,j} \in [-7, 7]$ .

**Table I.** Generation of  $\Pi$  multiples as high and low restricted DSSD numbers

$\mu_i$	$\mu_i$		$-\mu_i$		$3\mu_i$		$-3\mu_i$		$4\mu_i$		$-4\mu_i$	
	$h_i^1$	$l_i^1$	$h_i^{-1}$	$l_i^{-1}$	$h_i^3$	$l_i^3$	$h_i^{-3}$	$l_i^{-3}$	$h_i^4$	$l_i^4$	$h_i^{-4}$	$l_i^{-4}$
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	-1	0	3	0	-3	0	4	0	-4
2	0	2	0	-2	1	-4	-1	4	1	-2	-1	2
3	0	3	0	-3	1	-1	-1	1	1	2	-1	-2
4	0	4	0	-4	1	2	-1	-2	2	-4	-2	4
5	0	5	0	-5	1	5	-1	-5	2	0	-2	0
6	1	-4	-1	4	2	-2	-2	2	2	4	-2	-4
7	1	-3	-1	3	2	1	-2	-1	3	-2	-3	2

Therefore, computation of the two components of the DSSD digits of the  $j^{\text{th}}$  partial product (i.e.,  $p_{i,j}$  and  $p'_{i,j}$  in Eqn. set 14) does not produce any carry. This is also shown by Fig. 7 and Eqn. set 15, where  $l, h$  and  $p$  digits are represented by  $L_3l_2l_1l_0, H_2h_1h_0$  and  $P_3p_2p_1p_0$ , respectively and  $c_2 = l_1h_1 \vee l_1l_0h_0 \vee h_1l_0h_0$  is the carry into position 2.

$$\begin{aligned} p_0 &= l_0 \oplus h_0, p_1 = l_1 \oplus h_1 \oplus (l_0h_0), p_2 = l_2 \oplus H_2 \oplus c_2, \\ p_3 &= \overline{c_2}(L_3 \vee H_2\overline{l_2}) \vee L_3\overline{l_2} \vee L_3H_2 \end{aligned} \quad (15)$$



**Figure 7.** Addition of the Low and high components.

The latency of the described 13-logic-level PPG is evaluated as 9.33 FO4 via Logical effort analysis [23].

## 6. Fully redundant DSSD divider

In this section we briefly examine the impact of fully redundant operands and results on the state of the art decimal hardware division algorithms, whether multiplicative or subtractive, where  $Z, D, Q$  and  $R$  denote dividend, divisor, quotient and remainder, respectively.

### 6.1. DSSD multiplicative division

The reciprocal of the divisor  $1/D$  is produced either via converging multiplications, as in Eqn. 16, or

approximation of  $1/D$  with the help of a table lookup operation (Eqn. 17).

$$R_{j+1} = R_j(2 - R_jD) \quad (16)$$

The recurrence Eqn. 16 is specifically suitable for binary division, where  $D$  and  $R_j$  are fractions,  $R_0$  assumes an initial approximation of  $1/D$ , and each iteration is composed of two multiplications before and after a two's complement operation [27].

The other multiplicative method [28] is based on Eqn. (17), where again  $D$  is a normalized fraction and  $D_h$  and  $D_l$  are its most- and least-significant halves, respectively.  $D_h - D_l$  is simply obtained by aligning  $-D_l$  on the right of  $D_h$ ; thus no subtraction is actually required.  $D_h^{-2}$  can be looked up in a table, in parallel with the multiplication  $Z \times (D_h - D_l)$ . Finally, a multiplication by  $D_h^{-2}$  derives the quotient. A similar method using approximation based on Taylor series expansion is offered in [29], where the designer decides on where to break  $D$  into two parts.

$$\frac{Z}{D} = \frac{Z}{(D_h + D_l)} = \frac{Z(D_h - D_l)}{(D_h^2 - D_l^2)} \cong Z(D_h - D_l)D_h^{-2} \quad (17)$$

We believe that Eqn. 17, is more suitable for DSSD division than Eqn. 16. The reason is that no decimal subtraction is needed here. The negation required for  $-D_l$  can be performed by 4-bit two's complementation of all the DSSD digits in parallel with a latency of two logic levels. The multiplier design of the previous section can be used for the required two multiplications.

**Table II:** Logical equations for the bits of  $h_i^m$  and  $l_i^m$ .

Position		3	2	1	0
$\mu_i$	$l_i^1$	$\mu_i^2 \mu_i^1$	$\mu_i^2$	$\overline{\mu_i^2} \mu_i^1$	$\mu_i^0$
	$h_i^1$		0	0	$\mu_i^2 \mu_i^1$
$-\mu_i$	$l_i^{-1}$	$\overline{\mu_i^2} \mu_i^0 \vee \overline{\mu_i^2} \mu_i^1 \vee \mu_i^2 \overline{\mu_i^1}$	$\overline{\mu_i^2} \mu_i^0 \vee \overline{\mu_i^2} \mu_i^1 \vee \mu_i^2 \mu_i^0$	$\overline{\mu_i^1} \mu_i^0 \vee \mu_i^2 \mu_i^0 \vee \mu_i^2 \mu_i^1 \mu_i^0$	$\mu_i^0$
	$h_i^{-1}$		$\mu_i^2 \mu_i^1$	$\mu_i^2 \mu_i^1$	$\mu_i^2 \mu_i^1$
$3\mu_i$	$l_i^3$	$\overline{\mu_i^2} \mu_i^1 \vee \mu_i^1 \mu_i^0$	$\overline{\mu_i^2} \mu_i^1 \vee \mu_i^1 \mu_i^0 \vee \mu_i^2 \mu_i^1 x_0$	$\mu_i^2 \oplus \mu_i^0$	$\mu_i^0$
	$h_i^3$		0	$\mu_i^2 \mu_i^1$	$\mu_i^2 \oplus \mu_i^1$
$-3\mu_i$	$l_i^{-3}$	$\mu_i^2 \mu_i^0 \vee \mu_i^2 \mu_i^1 \vee \mu_i^1 \mu_i^0$	$\mu_i^2 \oplus \mu_i^1 \oplus \mu_i^0$	$\mu_i^2$	$\mu_i^0$
	$h_i^{-3}$		$\mu_i^2 \vee \mu_i^1$	$\mu_i^2 \vee \mu_i^1$	$\mu_i^2 \oplus \mu_i^1$
$4\mu_i$	$l_i^4$	$\overline{\mu_i^2} \mu_i^1 \mu_i^0 \vee \mu_i^2 \mu_i^1 \mu_i^0 \vee \mu_i^2 \mu_i^1 \mu_i^0$	$\mu_i^2 \mu_i^1 \vee \mu_i^2 \mu_i^0 \vee \mu_i^1 \mu_i^0 \vee \mu_i^2 \mu_i^1 \mu_i^0$	$\overline{\mu_i^2} \mu_i^1 \vee \mu_i^1 \mu_i^0$	0
	$h_i^4$		0	$\mu_i^2$	$\overline{\mu_i^2} \mu_i^1 \vee \mu_i^1 \mu_i^0$
$-4\mu_i$	$l_i^{-4}$	$\overline{\mu_i^2} \mu_i^0 \vee \mu_i^2 \mu_i^1 \mu_i^0$	$\mu_i^2 \oplus \mu_i^0$	$\overline{\mu_i^2} \mu_i^1 \vee \mu_i^1 \mu_i^0$	0
	$h_i^{-4}$		$\mu_i^2 \vee \mu_i^1$	$\mu_i^2 \vee \mu_i^1$	$\overline{\mu_i^2} \mu_i^1 \vee \mu_i^1 \mu_i^0$

## 6.2. DSSD subtractive division

The subtractive method is typically based on Eqn. 18, where  $W_0 = Z < D$ ,  $W_j$  is the partial remainder after  $j$  iterations,  $Z$  ( $D = .d_1d_2\dots d_m$ ) is normally a  $2m$  ( $m$ ) digit fraction with  $d_1 \neq 0$ , and  $q_{j+1}$  is the  $(j+1)^{\text{th}}$  digit of the fractional quotient  $Q = .q_1q_2\dots q_m$ .

$$W_{j+1} = 10W_j - q_{j+1}D \quad (18)$$

A radix-10 subtractive (or digit recurrence) division method has been recently proposed in [8] for nonredundant decimal operands, where quotient digits are primarily produced as  $[-7, 7]$  digits (i.e., DSSD) and converted to conventional decimal, on the fly. In the fully redundant DSSD division that we propose, the dividend, divisor, quotient and the partial remainders are all represented as DSSD numbers. We can implement Eqn. 18 using the multiplier of the previous section, for precomputation of DSSD multiples of the divisor (i.e.,  $\{-7D, -6D, \dots, 6D, 7D\}$ ), and the subtractor of Section 4, for producing the partial remainders. We adopt the techniques described in [8], for the DSSD representation, to obtain the comparison multiples and new quotient digits. For the former a look up table can be set up based on the three most significant digits (MSD) of  $D$  and for the latter the three MSDs of partial remainders are kept as a binary number (the rest in original DSSD form) to allow the use of very fast binary carry-save adders in deriving the three MSDs of the new partial remainder. Moreover, it can be shown that as is the case in [8] the three MSDs of the difference of the partial remainder and a comparison multiple are sufficient for quotient digit selection. The only extra units that are to be specifically designed for our fully redundant division scheme are the DSSD-to/from-binary converter of the three MSDs and the sign detector for the three MSDs of the comparison differences.

## 7. Comparison

It was noted at the end of Section 3 that the proposed DSSD adder is twice as fast, in terms of logic levels, as compared to the best previous work in [13]. This rough comparison was supported by evaluation of FO4 delays of the two circuits with 7.80 FO4 for the proposed design versus 13.56 FO4 of the previous one. To assess this advantage in a more realistic manner, and also for area comparison, we produced VHDL code for all the redundant decimal adders that we have encountered in the literature and synthesized them using the Synopsis Design Compiler. The target library was based on TSMC 0.13  $\mu\text{m}$  standard CMOS technology. The results appear in Table III, where the proposed design outperforms all the previous ones.

**Table III.** delay/area for SD decimal adders

Adder; ref.	Digit set	Delay (ns)	Ratio	Area ( $\mu\text{m}^2$ )	Ratio
Svoboda; [12]	$[-6, 6]$	2.50	2.87	781	1.25
RBCD; [13]	$[-7, 7]$	1.22	1.40	668	1.07
DSD; [14]	$[-9, 9]$	1.39	1.60	2333	3.75
DSD; [30]	$[-9, 9]$	1.65	1.90	2112	3.39
DSSD; new	$[-7, 7]$	0.87	1	622	1

We have not encountered any fully redundant decimal multiplier or divider in the literature to serve as a comparison basis with our designs. However, the fastest reported nonredundant decimal parallel multiplier is due to [6] with 65 FO4 delay, while the latency of the proposed fully redundant one is evaluated to be 48.33 FO4. As for the fully redundant DSSD divider, proposed in Section 6, we believe that the latency will not be more than the nonredundant radix-10 divider of [8]. The reason, in brief, is that our design basically mimics the nonredundant one, where the quite similar quotient digit selection of both designs is in the critical delay path of each recurrence.

## 8. Conclusions

We have proposed a framework for fully redundant decimal arithmetic based on decimal septa signed digit (DSSD) set  $[-7, 7]$ , where the operands and results of the four basic operations are represented as DSSD numbers. This allows for ultra fast carry-free addition and subtraction of DSSD numbers, which can lead to considerable speed up of conventional general computations, where the frequency of add/subtract operations is more than that of multiplication and division. To avoid conversion of DSSD result to conventional decimal, except at the end of computation, we proposed fully redundant DSSD multipliers and dividers to complete the required framework. Contrary to binary or high power-of-two radix redundant arithmetic, no extra storage is required for representation of DSSD numbers in comparison with conventional nonredundant decimal arithmetic. Therefore, a whole computation can be conducted efficiently, with the high speed gained due to highly frequent carry-free additions/subtractions, within the proposed DSSD framework up to the point of reporting a result to an output device.

The detailed designs for the proposed DSSD adder, subtractor, and unified add/subtract unit were explained. The results of synthesis of the proposed adder and other previous adders, as was tabulated in Table III, show 40% speed improvement over the fastest previous design. The proposed fully redundant parallel decimal multiplier, apparently the first one of this kind, is compared with the best nonredundant multiplier due to [6] showing 34% speed improvement.

We believe that the latency of the proposed fully redundant DSSD divider is no more than that of the nonredundant one due to [8]. The same outline design can be applied for a fully redundant DSSD square rooter. This research can be further continued by synthesis of the proposed multiplier and divider and using benchmarks to evaluate the speed advantage of the proposed framework in typical computations.

## Acknowledgement

This research has been funded in part by the IPM School of Computer Science under grant #CS1387-3-01 and in part by Shahid Beheshti University under grant #D/600/212.

## References

- [1] Cowlshaw, M. F., "Decimal Floating-Point: Algorithm for Computers," in *Proc. of the 16<sup>th</sup> IEEE Symposium on Computer Arithmetic*, pp. 104-111, Jun. 2003.
- [2] Busaba, F. Y., Krygowski C. A., Li W. H., Schwarz E. M., and Carlough S. R., "The IBM z900 Decimal Arithmetic Unit," in *Proc. of the 35<sup>th</sup> Asilomar Conference on Signals, Systems, and Computers*, Vol. 2, pp. 1335-1339, Nov. 2001.
- [3] Shankland, S., "IBM's POWER6 Gets Help with Math, Multimedia," ZDNet News, Oct. 2006.
- [4] Webb C. F., "IBM z10: The Next-Generation Mainframe Microprocessor," *IEEE Micro*, Vol. 28, Issue 2, pp. 19-29, 2008.
- [5] Lang, T. and A. Nannarelli, "A Radix-10 Combinational Multiplier," in *Proc. of the 40<sup>th</sup> Asilomar Conference on Signals, Systems, and Computers*, Nov. 2006.
- [6] Vazquez, A., E. Antelo, and P. Montuschi, "A New Family of High-Performance Parallel Decimal Multipliers," in *Proc. of the 18<sup>th</sup> IEEE Symposium on Computer Arithmetic*, pp. 195-204, 2007.
- [7] Nikmehr H., B. Phillips, and C.-C. Lim, "Fast Decimal Floating-Point Division," *IEEE Trans. on VLSI Systems*, Vol. 14, pp.951-961, 2006.
- [8] Lang T. and, A. Nannarelli, "A Radix-10 Digit-Recurrence Division Unit: Algorithm and Architecture," *IEEE Trans. Computers*, 56(6), pp. 727-739, Jun. 2007.
- [9] Vazquez A., E. Antelo, P. Montuschi, "A Radix-10 SRT Divider Based on Alternative BCD Codings," in *Proc. of the 25<sup>th</sup> International Conference on Computer Design*, pp. 280-287, Oct. 2007.
- [10] Jimeno A., Higinio Mora, Jose L. Sanchez, and Francisco Pujol, "A BCD-Based Architecture for Fast Coordinate Rotation," *Journal of System Architecture*, Vol. 54, Issue 8, pp. 829-840, Aug. 2008.
- [11] Crookes D., and M. Jiang, "Using Signed Digit Arithmetic for Low-power Multiplication," *Electronics Letters*, Vol. 43, No. 11, pp. 613-614, May 2007.
- [12] Svoboda, A., "Decimal Adder with Signed Digit Arithmetic," *IEEE Trans. on Computers*, Vol. C-18, No. 3, pp. 212-215, Mar. 1969.
- [13] Shirazi, B., D.Y. Yun, and C.N. Zhang, "RBCD: Redundant Binary Coded Decimal Adder," in *IEE Proc. Computer & Digital Techniques*, Vol.36, No.2, Mar. 1989.
- [14] Nikmehr H., B. J. Phillips, and C. C. Lim, "A Decimal Carry-Free Adder," in *Proc. of the SPIE Conference. Smart Mater., Nano-, Micro-Smart Syst.*, pp. 786-797, Dec. 2004.
- [15] Erle, M. A., E. M. Schwartz, and M. J. Schulte, "Decimal Multiplication with Efficient Partial Product Generation," in *Proc. of the 17<sup>th</sup> IEEE Symposium on Computer Arithmetic*, pp. 21-28, Jun. 2005.
- [16] Oberman S. F., *Design Issues in High Performance Floating Point Arithmetic Units*. PhD thesis, Stanford University, Nov. 1996.
- [17] Kenney, R. D., M. J. Schulte and M. A. Erle, "A High-Frequency Decimal Multiplier," in *Proc. of the IEEE International. Conference on computer Design: VLSI in Computers and Processors*, pp. 26-29, Oct. 2004.
- [18] Schmookler, M. and Weinberger A., "High Speed Decimal Addition," *IEEE Trans. on Computers*, Vol. C-20, No. 8, pp. 862-866, Aug. 1971.
- [19] Vazquez A. and E. Antelo, "Conditional Speculative Decimal Addition," in *Proc. of the 7<sup>th</sup> Conference on Real Numbers and Computers*, pp. 47-57, Jul. 2006.
- [20] Parhami, B., "Generalized Signed-Digit Number System: A Unifying Framework for Redundant Number Representation," *IEEE Trans. on Computer*, Vol. 39, No. 1, pp. 89-98, 1990.
- [21] M. Daumas, and D. W. Matula, "Further Reducing the Redundancy of a Notation over a Minimally Redundant Digit Set," *Journal of VLSI signal processing*, Vol. 33, pp. 7-18, 2003.
- [22] Kornerup Peter, "Reviewing 4-to-2 Adders for Multi-Operand Addition," *Journal of VLSI Signal Processing*, Springer, Vol. 40, No. 1, pp. 143-152, May 2005.
- [23] Sutherland, I. E., R. F. Sproull and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufman, 1999.
- [24] Dadda, L., "Multi Operand Parallel Decimal Adder: a Mixed Binary and BCD Approach," *IEEE Trans. on Computers*, Vol. 56, No. 10, pp. 1320-1328, Oct. 2007.
- [25] Erle, M. A. and M. J. Schulte, "Decimal Multiplication Via Carry-Save Addition," in *Proc. of the Conference on Application-Specific Systems, Architectures, and Processors*, pp. 348-358, Jun. 2003.
- [26] Jaberipur G. and Amir Kaivani, "Improving the Speed of Parallel Decimal Multiplication," *IEEE Trans. on Computers*, to appear.
- [27] Ercegovac M. D., and T. Lang, *Digital Arithmetic*, Morgan Kaufman, 2004.
- [28] Flynn M. J., and S. F. Oberman, *Advanced Computer Arithmetic Design*, John Wiley, 2001.
- [29] Wang L. K., and M. J. Schulte, "A Decimal Floating-Point Divider Using Newton-Raphson Iteration," *Journal of VLSI Signal Processing*, Vol. 49, pp. 3-18, 2007.
- [30] Moskal, J., E. Oruklu, J. Saniie, "Design and Synthesis of a Carry-Free Signed-Digit Decimal Adder," in *Proc. of the IEEE International Symposium on Circuits and Systems*, pp. 1089-1092, May 2007.