Efficient Arithmetic Implementations Based on Carry-Save Representations

Dhananjay S. Phatak[a], Tom Goff[a] and Israel Koren[b]

[a] Electrical Engr. Dept. State Univ. of New York
Binghamton, NY 13902-6000, U.S.A.

[b] Electrical and Computer Engr. Dept.
Univ. of Massachusetts, Amherst, MA 01003, U.S.A.

ABSTRACT

This paper presents arithmetic implementations which use binary redundant numbers based on carry-save representations. It is well-known that constant-time addition, in which the execution delay is independent of operand length, is feasible only if the result is expressed in a redundant representation. Carry-save based formats are one type of a redundant representation which can lead to highly efficient implementations of arithmetic operations. In this paper, we discuss two specific carry-save formats that lead to particularly efficient realizations. We illustrate these formats, and the "equal-weight grouping" (EWG) mechanism wherein bits having the same weight are grouped together during an arithmetic operation. This mechanism can reduce the area and delay complexity of an implementation. We present a detailed comparison of implementations based on these two carry-save formats including measurements from VLSI cell layouts. We then illustrate the application of these VLSI cells for multi-operand additions in fast parallel multipliers. Finally, we also indicate the relationship with previous results. [1]

Keywords: Redundant number systems, carry-save, VLSI cells, addition.

1. INTRODUCTION

A positional radix-β number system represents an n-digit value V as a string of digits, $(d_{n-1}, d_{n-2}, \cdots d_0)$, where

$$\sum_{i=0}^{n-1} d_i \cdot \beta^i = V$$

The value that each digit, d_i, can assume is determined by the digit set for that position D_i, such that $d_i \in D_i$. In conventional representations, the digit set is the same for all positions and is defined by $D = \{d | 0 \leq d \leq \beta - 1\}$. A number system is redundant if there is some value which does not have a unique representation. In other words, there exists an n-digit number which satisfies

$$\sum_{i=0}^{n-1} d_i \cdot \beta^i = \sum_{i=0}^{n-1} d_i' \cdot \beta^i, \qquad d_i, d_i' \in D_i$$

and there is some position j where $d_j \neq d_j'$. This implies that there is at least one digit position for which the cardinality of the set D_i satisfies $|D_i| > \beta$. We call such a position, a *redundant digit position*.

Addition can be thought to be an instance of the digit set conversion problem. [2–4] In this context, we consider the addition of two operands X and Y yielding the result $Z = X + Y$. The digit set $D_i^x + D_i^y$ can be thought of as the input digit set for position i, and D_i^z as the output digit set. Given this, addition is then the operation of converting from one digit set to another. In most cases the range of $D_i^x + D_i^y$ is larger than D_i^z making carry propagation necessary. This results in the carry relationship

$$z_i + \beta \cdot c_i = x_i + y_i + c_{i-1} \tag{1}$$

where c_{i-1} is the carry-in to position i, c_i is the carry-out, and both are members of a carry set C.

An alternate treatment of addition based on digit-set operations can be found in [1] which provides a framework for designing adders based on contiguous sets.

E-mails: phatak@ee.binghamton.edu, tomgoff@ibm.net, koren@ecs.umass.edu

Advanced Signal Processing Algorithms, Architectures, and Implementations X, Franklin T. Luk, Editor, Proceedings of SPIE Vol. 4116 (2000) © 2000 SPIE · 0277-786X/00/$15.00

1.1. Constant-Time Addition

Constant-time addition is possible at a redundant digit position i if the value of c_i can be determined by considering only a fixed number of previous input digits, making it independent of c_{i-1}. The number of previous digits required constitutes a right context, or look-back[2–4] and is henceforth denoted by L. The operation of constant-time addition at redundant digit positions can be explained conceptually as the two-step process described below.

Step 1: Based on its fixed right context, every redundant digit position generates an intermediate sum σ_i and an intermediate carry-out c_i, where

$$\sigma_i + \beta \cdot c_i = x_i + y_i = \theta_i \tag{2}$$

In other words, Equation (2) expresses the sum of operand digits $\theta_i = x_i + y_i$ as the pair (c_i, σ_i).

Step 2: The final sum z_i is formed by $z_i = \sigma_i + c_{i-1}$ where $\sigma_i + c_{i-1} \in D_i^z$.

If there are non-redundant digit positions in the result Z, carries must ripple through them [5,6] and they are determined by (1).

As described in[2–4] , the carry-out of a digit position can be dependent on the input operands and the output digit set at that position, as well as those (operands and output digit sets) at all the digit positions that fall within the fixed-length right and left contexts. If a left context is actually used, this means that the carry-in to some position can be dependent on the input digits at that position.

1.2. Radix-2 Redundant Representations

Radix-2 representations are the most commonly used, and hence are fundamental. Several radix-2 redundant digit sets can be used, these are summarized below:

$$D^{(SD)} = \{-1, 0, 1\} \tag{3}$$
$$D^{(SD3^{\langle - \rangle})} = \{-2, -1, 0, 1\} \tag{4}$$
$$D^{(SD3^{\langle + \rangle})} = \{-1, 0, 1, 2\} \tag{5}$$
$$D^{(CS2)} = \{0, 1, 2\} \tag{6}$$
$$D^{(CS3)} = \{0, 1, 2, 3\} \tag{7}$$

Note that a redundant binary digit needs at least two bits to represent it. In fact, all the redundant digit sets listed above need exactly two bits to represent their digit values. In essence, several redundant representations where some digit positions have two bits allocated to them can accomplish constant time addition and simultaneous format conversion. The interesting question is given this basic redundancy (i.e., redundant digits), which number representations lead to the most efficient implementations and best exploit the redundancy made available by the extra bits?

A detailed theoretical analysis of constant time addition and simultaneous format conversion in number representations based on the above digit sets[6,7] answers this question, showing that the carry-save representation it the best for fast and efficient implementation of word-parallel operations.[6,7] In general, two types of number systems based on each of these digit sets are possible: a fully redundant system and a partially-redundant system. A fully redundant system is one in which all digit positions of a number are redundant and the characteristics of such systems are well known.[8–11] In a partially redundant system only some digit positions are redundant and further details about such systems can be found in.[5–7]

In this paper we concentrate on two fully redundant (i.e., each digit position is redundant) number systems based on the carry-save representation, viz., the digit sets $D^{(CS2)}$ and $D^{(CS3)}$. We illustrate these formats, and the "equal-weight grouping" (EWG) mechanism wherein bits having the same weight are grouped together during an arithmetic operation. We present a detailed comparison of implementations based on these two carry-save formats including measurements from VLSI cell layouts. We then illustrate applications which use these VLSI cells as building blocks.

We also show connections with prior work[1] which does not address constant-time addition using the *CS*2 number system (see Section 5).

1.3. Redundant Binary Encodings

The number systems of interest (based on digit sets $D^{(CS2)}$ and $D^{(CS2)}$) need two bits to represent each redundant digit. While many encodings of two bits can be used to denote digit values, specific encodings lend themselves to efficient implementations.[6,7] Consider the encoding of an operand X as $(\hat{x}_{n-1},\hat{x}_{n-2},\cdots\hat{x}_0)$, where $\hat{x}_i$ is the radix-2 redundant digit in the i-th position (a hat notation, $\hat{x}_i$, indicates that the i-th digit of X is redundant and is encoded using two bits). The bits representing a redundant digit $\hat{x}_i$ can be thought of as having *higher* and *lower* significant bits (x_i^h, x_i^l), respectively. Note that arbitrary bit combinations can be used to represent redundant digit values, but we concentrate on encodings that satisfy the relationship

$$\text{value of digit } \hat{x}_i \ = \pm 2 \cdot x_i^h \pm x_i^l \tag{8}$$

It will be shown that such encodings lead to efficient implementations.

In carry-save redundant representations, following the literature, we refer to the higher significant bit, x_i^h, as the *carry* bit and the lower significant bit, x_i^l, as the *sum* bit. Here, the sum bit has a weight of 1 and the carry bit has a relative weight of $+2$. Thus, $\hat{x}_i = 2 \cdot x_i^h + x_i^l$, which yields the following encoding:

$$D^{(CS3)}: \quad (0,0) \equiv 0, \ \ (0,1) \equiv 1, \ \ (1,0) \equiv 2, \ \ (1,1) \equiv 3 \tag{9}$$

$D^{(CS2)}$ does not include the digit 3 which makes the bit pattern $(x_i^h, x_i^l) = (1,1)$ invalid for the *CS*2 representation.

1.4. Equal-Weight Grouping

The encoding of both signed-digit and carry-save redundant digits ensures that x_i^l and x_{i-1}^h have the same weight, i.e., the digits $\hat{x}_i$ and $\hat{x}_{i-1}$ *overlap* each other. This overlap provided by the chosen encodings can be exploited to reduce the range of digit sums that must be generated, and to predict the range of an incoming carry when two numbers are added. Figure 1 shows two redundant digits, $\hat{x}_i$ and $\hat{x}_{i-1}$, of a number X drawn as squares. The arrows are used to indicate the individual bits that make up each digit. The bits (x_i^l and x_{i-1}^h), both have a weight of 2^i and can be grouped to form an alternate interpretation of the bits. Instead of having digits of the form $\hat{x}_i = 2 \cdot x_i^h + x_i^l$ the bits can create "Equal-Weight-Grouped" (EWG) digits of the form $\hat{x}_i' = x_i^l + x_{i-1}^h$, without affecting the value of the original operand X.

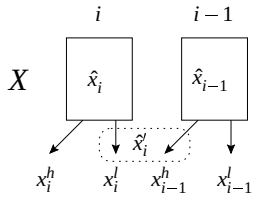

Figure 1. Equal-weight grouping.

To illustrate the impact of equal-weight grouping, consider adding the digits of two *CS*3 (i.e., conventional carry-save format) numbers X and Y, where the digit set is $D^{(CS3)} = \{0,1,2,3\}$. Normally the digit sum $\theta_i = 2 \cdot x_i^h + x_i^l + 2 \cdot y_i^h + y_i^l$ would be in the range $0 \le \theta_i \le 6$ which must be expressed as a final sum $0 \le z_i \le 3$ and a carry-out, c_i, which would be larger then 1. If EWG digits are added instead, the digit sum $\theta_i' = x_i^l + x_{i-1}^h + y_i^l + y_{i-1}^h$ is restricted to the range $0 \le \theta_i' \le 4$, which is still expressed with a final sum of $0 \le z_i \le 3$ but the carry-out, c_i, will be at most 1. As a result, the number of values needed for the carry-out is reduced.

Another benefit of working with bits originally belonging to distinct digits arises when considering digit sets which exclude some bit patterns, as in *CS*2. In these cases, the higher-significant bits from the less-significant digits, x_{i-1}^h and y_{i-1}^h, provide some information about what range the less-significant intermediate sum, θ_{i-1}, is in and therefore the range of the incoming carry. Note however that in these cases, the range of the intermediate sum is not affected by the equal-weight grouping.

Figure 2 illustrates the operation of the previously described constant-time addition process with the result expressed in a fully-redundant form. At position i, the EWG digits $\hat{x}_i'$ and $\hat{y}_i'$ are added together to form the group sum θ_i. Based on θ_i and the

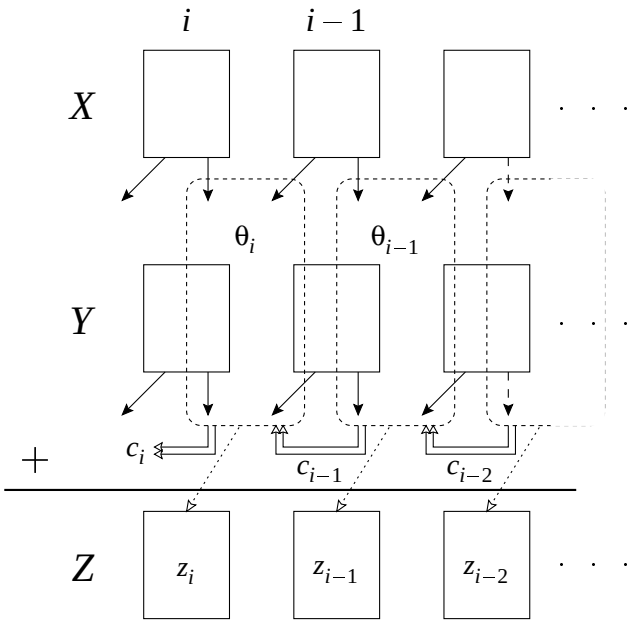

Figure 2. Constant-time addition without format conversion.

previous group sums $(\theta_{i-1}, \theta_{i-2}, \cdots)$ that make up the right context of position *i*, a carry-out c_i and intermediate sum σ_i which satisfy (2), are chosen. The final sum z_i is formed by adding σ_i and c_{i-1}, the carry-in from the previous position.

Given this framework for constant-time addition, we consider next the specific instances of the redundant radix-2 fully redundant number systems based on digit sets $D^{(CS2)}$ and $D^{(CS3)}$. In Sections 2 and 3, we discuss the rules for constant-time addition of redundant Carry-Save format numbers. In Section 4 we compare the two number systems. We then show VLSI implementations of adder cells and present the corresponding cell delays. We also discuss parallel multipliers based on these two representations. Section 5 presents a discussion of some theoretical issues and conclusions.

2. *CS*2 ADDITION

This section considers *CS*2 constant-time addition. The digit set at each redundant position is $D^{(CS2)} = \{0,1,2\}$, and as mentioned earlier, the encoding prevents the bit combination $(x_i^h, x_i^l) = (1,1)$ from occurring. To find the carry values needed, note that the sum of EWG bits of weight 2^i must be expressible in terms of a sum digit and a carry-out:

$$\theta_i + c_{i-1} = x_i^l + y_i^l + x_{i-1}^h + y_{i-1}^h + c_{i-1} \quad \leq \quad z_{i_{\max}} + c_i \cdot 2 \quad \text{where} \quad z_{i_{\max}} = 2 \tag{10}$$

Without restricting x_i^l and y_i^l, the carry-out can be limited to $c_i \in \{0,1\}$ if

$$x_{i-1}^h + y_{i-1}^h + c_{i-1} \quad \leq \quad 2 \tag{11}$$

If one or both bits x_{i-1}^h and y_{i-1}^h are 0 then condition (11) is obviously satisfied. The only case that needs further scrutiny is when both $x_{i-1}^h = y_{i-1}^h = 1$. In this case it can be shown that the carry c_{i-1} must be 0 as summarized by the following lemma.

Lemma 1: For the *CS*2 encoding, if $x_{i-1}^h \cdot y_{i-1}^h = 1$ then $c_{i-1} = 0$. In other words, if the upper bits of an EWG digit are both 1, there will be no carry-in to that place.

Proof:

Denote $\theta_i^h = x_{i-1}^h + y_{i-1}^h$. Because of the *CS*2 encoding, $\theta_i^h = 2$ implies $x_{i-1}^l = y_{i-1}^l = 0$ and as a result $\theta_{i-1} \in \{0,1,2\}$. $\theta_{i-1} \in \{0,1\}$ will never produce a carry-out of 2. The only concern is when $\theta_{i-1} = 2$, which in turn,

implies $\theta_{i-2} \in \{0,1,2\}$. Consequently, position $i-1$ could produce a carry-out of 2 only if $\theta_{i-2} = 2$. This is now a recursive/circular argument since position $i-2$ can produce a carry-out of 2 only if $\theta_{i-3} = 2$, and so on. Since all the numbers being considered are assumed to be of some fixed length, the question becomes; can a string of intermediate sums $\theta_i^h\ \theta_{i-1}\ \theta_{i-2}\ \cdots\ \theta_{i-w} = 2\ 2\ 2\ \cdots\ \theta_{i-w}$ terminate with $\theta_{i-w} > 2$. Such a string can only occur when $\theta_{i-1}^h\ \theta_{i-2}^h\ \cdots\ \theta_{i-w+1}^h = 2\ 2\ \cdots\ 2$ which means that $(x_{i-w}^h, y_{i-w}^h) = (1,1)$ which in turn, implies that $(x_{i-w}^l, y_{i-w}^l) = (0,0)$ (because of the *CS*2 encoding). This results in $\theta_{i-w} \in \{0,1\}$ (since the string of 2's terminates with θ_{i-w}, $\theta_{i-w} \neq 2$).

In essence, a string of EWG digit-sums of value 2 must eventually terminate in some position $i-w$ with $\theta_{i-w} \in \{0,1\}$. This means position $i-w$ can never produce a carry-out, therefore positions $i-w+1$ through i can leave their intermediate sums as $\sigma = 2$, thereby implying that $c_{i-1} = 0$ □

This means that the carry set $C^{(CS2)} = \{0,1\}$ is sufficient. Given this, the rules for *CS*2 addition without any format conversion can be simplified as shown in Table 1.

θ_i	$x_{i-1}^h + y_{i-1}^h$	c_i	σ_i
0	×	0	0
1	×	0	1
2	2	0	2
	otherwise	1	0
3	×	1	1
4	×	1	2

Table 1. Rules for *CS*2 Addition

Note that there is no need to look back at any previous digits, in other words, the (intermediate) carry-out of a position depends only on the sum of bits in that position. Hence the context or the look-back is $L = 0$.

It can be verified that if the equal weight grouping is not done, then a carry of value 2 is required, resulting in the carry-set $\{0,1,2\}$ which implies that two bits are needed to encode the carry if EWG is not employed. This demonstrates the advantage of the EWG scheme.

3. *CS*3 ADDITION

Here, every output digit position is redundant and can assume any of the values $\{0,1,2,3\}$. Since 3 is an allowable digit, the carry-relationship

$$\theta_{i_{\max}} + c_{i-1_{\max}} \leq z_{i_{\max}} + 2 \cdot c_{i_{\max}} \tag{12}$$

simplifies to $c_{i_{\max}} \geq 1$ (assuming $c_{i-1_{\max}} = c_{i_{\max}}$). This makes the carry set $C^{(CS3)} = \{0,1\}$ sufficient for *CS*3 addition without format conversion. The rules for determining σ_i and c_i are given in Table 2. Again, they are stated only in terms of θ_i, without any dependency on the previous group sum which makes the look-back $L = 0$.

Once again it can be verified that if the equal weight grouping is not utilized, then a carries of value 2 and 3 must also be allowed resulting in the carry-set $\{0,1,2,3\}$. Such a scheme would require two bits to encode the carry and will have more complex logic. Hence, it is advantageous to use the EWG scheme.

4. COMPARISON

Table 3 summarizes the look-back distances, L, and carry sets needed for the two types of redundant binary addition considered. The table clearly shows that equal-weight grouping can lead to smaller carry sets and a smaller look-back.

Table 3 shows that the minimum look-back occurs only when the proposed equal-weight-grouping is employed. In the table, the representation requiring the smallest carry-set should be selected because a smaller carry set usually implies less

θ_i	c_i	σ_i
0	0	0
1	0	1
2	1	0
	or	
	0	2
3	1	1
4	1	2

Table 2. *CS*3 Addition

Operation	Carry-Set		Look-Back L (Number of radix-2 digits)	
	Equal-Weight Grouping (EWG)	No EWG	EWG	No EWG
$CS2 + CS2 \rightarrow CS2$	$\{0,1\}$	$\{0,1,2\}$	0	1
$CS3 + CS3 \rightarrow CS3$	$\{0,1\}$	$\{0,1,2,3\}$	0	1

Table 3. Comparison of binary constant-time addition techniques.

complex logic which should translate into smaller area and critical path delay. Applying these criteria, it is seen that the *CS*2 and *CS*3 representations are roughly equivalent (same carry set and context with EWG). However, Table 3 compares the number representations only at an abstract level, in terms of the size of the carry-set and the look-back L. While this comparison can provide a good high-level assessment, actual VLSI implementations are necessary to gauge the relative merits and disadvantages of the redundant representations. In the next subsection, we show simulation measurements on VLSI layouts of adder cells for these two representations.

4.1. Implementation

In order to verify some of the comparison results included in Table 3, we designed (whenever required), laid out and simulated the VLSI adder cells for both the above representations. Note that the cell that implements $CS3 + CS3 \rightarrow CS3$ accepts four bits of equal weight as inputs (two sum bits and two carry bits, one from each of the two operands). It also accepts a carry-in from the adjacent position and generates an output in the carry-save format (i.e., two output bits) and a carry-out which is *independent* of the carry-in. This is nothing but a 4:2 compressor employed in conventional multipliers. The 4:2 compressor presented in[12] is extremely efficient and hence we laid out this compressor (For the sake of brevity, the gate diagram and details of this cell are omitted, those can be found in[12]).

The cell to implement $CS2 + CS2 \rightarrow CS2$ was newly designed. Its gate diagrams is shown in Figure 3. In the figure, it is seen that the carry-out is generated based only on the bits of the *current* group, i.e., there is no look-back.

VLSI layouts of both the cells were simulated in the TSMC SCN025 0.25 micron technology process (available from MOSIS) with a 2.5 volt supply. The designs were first verified at the logic level. Berkeley SPICE 3f5 was used to estimate the critical path delay of each cell, which included appropriate fan-in as well as fan-out loading for all components. The results are summarized in Table 4.

Adder Cell	Critical Path Delay (ns)
*CS*2	0.66100
*CS*3	0.46580

Table 4. The delay of the critical path through one cell from SPICE simulations.

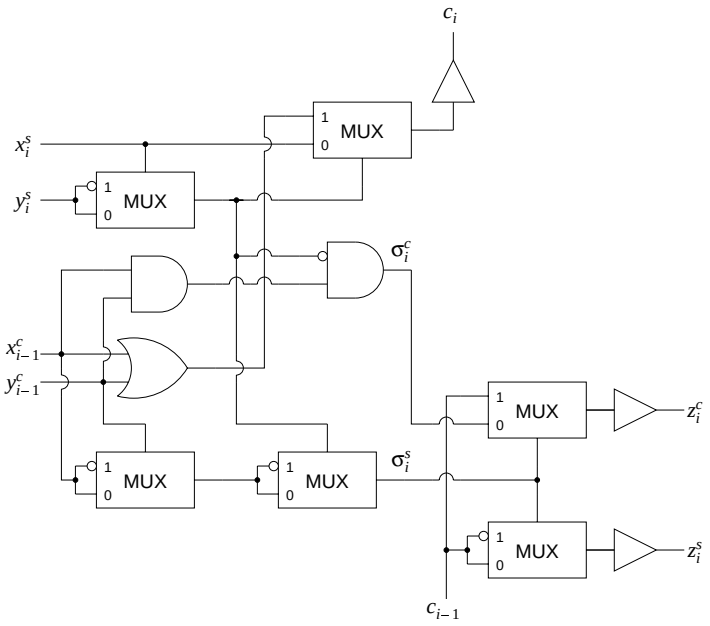

Figure 3. Cell for $CS2 + CS2 \rightarrow CS2$, i.e., a 4:2 compressor for the $CS2$ representation. $(x_i^l, y_i^l, x_{i-1}^h, y_{i-1}^h)$ are the four input operand bits of equal weight. The carry $c_i \in \{0,1\}$ needs a single bit line. $\sigma_i \in \{0,1,2\}$ is the intermediate sum encoded by the bits (σ_i^h, σ_i^l), with $\sigma_i = 2 \cdot \sigma_i^h + \sigma_i^l$. The output is encoded by (z_i^h, z_i^l) and can assume any of the three values $\{0,1,2\}$.

It should be noted that the SPICE simulation results are highly layout dependent. These layouts were done for a relative comparison of the redundant adder cells. Hence, as long as the same layout strategy was adopted in both designs, the *relative* comparison is informative. However such a "uniform" layout strategy rules out fine tuning; in particular, the $CS2$ cell could be made more compact which might have a significant impact on the overall delay.

In light of the above results, it can be seen that for a multiply operation, using the $CS3$ representation with the compressor presented in,[12] is likely to yield faster implementations. Note that converting partial products from two's complement format to $CS3$ format is trivial; it requires no gates at all. This is illustrated in Figure 4 which shows that merely grouping the bits appropriately leads to a valid output in the $CS3$ representation.

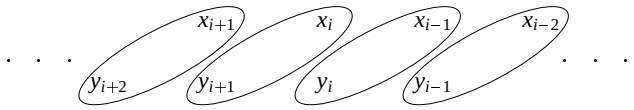

Figure 4. Combining two's complement operands to generate an output in $CS3$ format.

In contrast, if the $CS2$ (or conventional SD representation) is employed, two's complement partial products must be added to generate outputs in their respective formats. In each of these cases, a small delay worth about one full adder is required to achieve this conversion. For the $CS2$ format, this need is straightforward; a $1+1$ must be converted into a $(1,0)$, since the pattern $(1,1)$ is not allowed. Furthermore, the 4:2 compressor that performs $CS3 + CS3 \rightarrow CS3$ is smaller and faster than other cells. These two factors together imply that multipliers based on $CS3$ are expected to outperform multipliers based on the $CS2$ representation.

5. DISCUSSION AND CONCLUSIONS

In this section, we show the relationship of this work to some results presented in [1] and present concluding remarks. The examples of constant-time addition that we have described can be re-written using the notation from [1] as shown below.

$$CS2: \quad 2\langle 1^0\rangle + \langle 2^0\rangle \Leftarrow \langle 1^0\rangle + \langle 1^0\rangle + \langle 1^0\rangle + \langle 1^0\rangle + \langle 1^0\rangle$$
$$CS3: \quad 2\langle 1^0\rangle + \langle 3^0\rangle \Leftarrow \langle 1^0\rangle + \langle 1^0\rangle + \langle 1^0\rangle + \langle 1^0\rangle + \langle 1^0\rangle$$

The notation shows that the sum of digit sets to the right of the decomposition operator ($\Leftarrow$) is expressed using the digit sets to the left of the operator. A digit set $\langle \delta^{\omega} \rangle$ is characterized by its diminished cardinality, δ, and negative offset from zero, ω. This represents digits in the range $[-\omega, -\omega+\delta]$ and must include 0 (further details regarding the notation and decompositions can be found in[1]).

The analysis in[1] requires that the total diminished cardinality to the left of the decomposition operator, δ_{out}, be greater than or equal to the total diminished cardinality of the right side, δ_{in}. The condition that $\delta_{out} \geq \delta_{in}$ is satisfied by the *CS*3 representation. However, for the *CS*2 representation $\delta_{out} = 4$ and $\delta_{in} = 5$; violating the diminished cardinality condition. Therefore, *CS*2 addition presented in this paper lies outside the framework developed in. [1]

In conclusion, this paper presents a comprehensive analysis of constant-time addition based on the *CS*2 and *CS*3 redundant representations. We used the notion of "equal-weight grouping" (EWG), wherein bits having the same weight are grouped together during the constant-time addition operation. The analysis and data show that EWG leads to efficient implementations. We illustrated the relationship of our work to prior results in [1] and showed that the *CS*2 format considered here has interesting properties which transcend the framework for redundant representations presented in. [1]

Possible future work includes finding redundancy metrics which also capture the complexity of hardware implementations based on the redundant format under consideration. As shown in, [6,7] merely utilizing all possible combinations of bits available to encode a digit to represent distinct values does not necessarily lead to the same complexity in all cases. This needs further scrutiny. Another issue is to extend the necessary and sufficient conditions for constant-time addition (derived in [2]) to the case where the digit sets at all digit positions are not the same. Such a framework allows for arbitrary spacing of redundant digit positions throughout a representation, as well as the ability to vary the types of redundant digits used. It is conceivable that examples of situations where both left and right contexts are required could arise in such cases. Since the digit sets could be radically different from one digit position to the next, it is possible that each position needs to also examine the left context in order to select the appropriate/acceptable carry value which it can output.

Acknowledgement

We would like to thank Prof. Naofumi Takagi for his insightful remarks which led us to this investigation. We also wish to thank Professors Milos Ercegovac and Neil Burgess for their constructive suggestions which improved the quality of this paper.

REFERENCES

1. T. M. Carter and J. E. Robertson, "The Set Theory of Arithmetic Decomposition," *IEEE Transactions on Computers*, **C-39**, pp. 993–1005, August 1990.
2. P. Kornerup, "Necessary and Sufficient Conditions for Parallel and Constant Time Conversion and Addition," in *Proc. 14th IEEE Symposium on Computer Arithmetic*, pp. 152–156, IEEE Computer Society, April 1999.
3. P. Kornerup, "Digit-Set Conversions: Generalizations and Applications," *IEEE Transactions on Computers*, **C-43**, pp. 622–629, May 1994.
4. A. M. Nielsen and P. Kornerup, "Redundant Radix Representation of Rings." *IEEE Transactions on Computers*, **C-48**, pp. 1153-1165, November 1999.
5. D. S. Phatak and I. Koren, "Hybrid Signed–Digit Number Systems: A Unified Framework for Redundant Number Representations with Bounded Carry Propagation Chains," *IEEE Trans. on Computers, Special issue on Computer Arithmetic*, **TC–43**, pp. 880–891, Aug. 1994. (Unabridged version available at http://www.ee.binghamton.edu/faculty/phatak).
6. T. Goff, D. S. Phatak, and I. Koren, "Redundancy Management in Arithmetic Processing via Redundant Binary Representations," *Proceedings of ASILOMAR'99 (Annual Conference on Signals Systems and Computers), Pacific Grove, California* , pp. 1475–1479, Oct. 1999.

7. D. S. Phatak, T. Goff, and I. Koren, "On Constant-time Addition and Simultaneous Format Conversion Based on Redundant Binary Representations," submitted for publication.
8. B. Parhami, "Generalized signed-digit number systems: a unifying framework for redundant number representations," *IEEE Transactions on Computers*, **C-39**, pp. 89–98, Jan. 1990.
9. C. Nagendra, R. M. Owens, and M. J. Irwin, "Unifying Carry-Sum and Signed-Digit Number Representations," Tech. Rep. CSE–96–036, Computer Science and Engineering Department, Pennsylvania State University, 1996.
10. I. Koren, *Computer Arithmetic Algorithms*, Brookside Court Publishers, Amherst, Massachusetts, 1998.
11. B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, Oxford University Press, 2000.
12. N. Ohkubo and Suzuki, M., et. al., "A 4.4-ns CMOS 54 $\times$ 54-b Multiplier Using Pass-Transistor Multiplexor," *IEEE Journal of Solid-State Circuits*, **30**, pp. 251–256, Mar. 1995.
13. S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, and N. Takagi, "Design of high speed MOS multiplier and divider using redundant binary representation," *Proc. of the 8th Symposium on Computer Arithmetic*, pp. 80–86, 1987.
14. J. J. J. Lue and D. S. Phatak, "Area $\times$ Delay ($A \cdot T$) Efficient Multiplier Based on an Intermediate Hybrid Signed–Digit (HSD–1) Representation," *Proc. of the 14th IEEE International Symposium on Computer Arithmetic, Adelaide, Australia*, pp. 216–224, April 1999.