Hybrid Signed–Digit Number Systems: A Unified Framework for Redundant Number Representations with Bounded Carry Propagation Chains

Dhananjay S. Phatak Electrical Engr. Dept.,
State University of New York
Binghamton, NY 1302–6000
phatak@ee.binghamton.edu

Israel Koren, *Fellow, IEEE*
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003

(IEEE Transactions on Computers, vol. 43, No. 8, August 1994 pp 880-891)

ABSTRACT

A novel hybrid number representation is proposed in this paper. It includes the two's complement representation and the signed-digit representation as special cases. The hybrid number representations proposed are capable of bounding the maximum length of carry propagation chains during addition to any desired value between 1 and the entire word length. The framework reveals a continuum of number representations between the two extremes of two's complement and signed-digit number systems and allows a unified performance analysis of the entire spectrum of implementations of adders, multipliers and alike.

We present several static CMOS implementations of a two–operand adder which employ the proposed representations. We then derive quantitative estimates of area (in terms of the required number of transistors) and the maximum carry propagation delay for such an adder. The analysis clearly illustrates the tradeoffs between area and execution time associated with each of the possible representations. We also discuss adder trees for parallel multipliers and show that the proposed representations lead to compact adder trees with fast execution times.

In practice, the area available to a designer is often limited. In such cases, the designer can select the particular hybrid representation that yields the most suitable implementation (fastest, lowest power consumption, etc.) while satisfying the area constraint. Similarly, if the worst case delay is predetermined, the designer can select a hybrid representation that minimizes area or power under the delay constraint.

Index terms : Bounded Carry Propagation, Carry–free addition, Hybrid Signed–Digit Number System, Redundant Number Representation, Signed–Digit Numbers, Static CMOS implementation.

I Introduction

The well known signed–digit (SD) number representation makes it possible to perform addition with carry propagation chains that are limited to a single digit position, and has been used to speed up arithmetic operations [1]–[7]. The SD representation also renders most-significant-digit-first schemes feasible and has been used in on–line arithmetic [3] and digit–pipelined schemes [4]. In the binary signed–digit number system, each digit can assume any one of the three values $\{-1, 0, 1\}$. As a result, redundancy is introduced in the number system, i.e., a number can be represented in more than one way. For example, 1 can be represented by 01 or $1\overline{1}$, where $\overline{1} = -1$. This redundancy can be exploited to limit the length of carry propagation chains to only one digit position [8], making it possible to add two numbers in fixed time, irrespective of the word length.

In the addition of two numbers represented in the conventional binary number system, on the other hand, the carry may propagate all the way from the least significant digit to the most significant. The addition time is thus dependent on the word length (linear in ripple–carry adders, logarithmic in carry–look–ahead adders). The speedup in addition time in the SD number system does not come without cost, however, since two bits are needed to represent a binary signed digit. Also, the basic adder cell that adds two signed digits and an input carry to produce a signed digit and a carry is more complex than a full adder for unsigned digits. Thus, more area is traded off for the constant addition time. The SD representation is especially useful for multi–operand addition. Signed–digit adder trees are easier to lay out and route than Wallace trees. In [5] a 64×64 bit multiplier based on a redundant signed–digit binary adder tree was shown to yield a smaller critical path delay than the corresponding Wallace tree multiplier. A similar design for a fast 54×54 bit multiplier was recently presented in [6]. Note that a full adder achieves a reduction from 3 to 2 operands. Hence, in a multi–operand adder tree composed of full adders, the reduction ratio achieved at each level is 3 to 2. In the addition of SD operands, on the other hand, the reduction ratio is 2 to 1 and this is one of the main advantages of using signed digits.

The SD and the two's complement number representations are at two extremes. In the SD number system more bits, switching devices and routing are required per digit. In return, the carry propagation is limited to a single digit position. In the conventional number systems on the other hand, fewer bits, switches and routing are needed per digit, but the carry propagates across the entire word length. We introduce a hybrid number representation where the maximum carry propagation length can be set to any desired value between one and the full word length. The area required decreases as the length of the carry propagation chain increases. Such a representation reveals a continuum of possible realizations that trade off area for speed. This framework also permits a unified analysis of the performance (in terms of area (A), execution time (T), power consumption, etc.) of the whole spectrum of implementations of adders, multipliers and the like.

The proposed number representation can be useful in practice when the area available to a designer is limited or the worst case delay is predetermined. If the area available is limited, the designer can select the particular hybrid representation that yields the most suitable implementation, i.e., the one

with the least delay or power consumption, under the area constraint. Conversely, given a worst case delay, the designer can select a hybrid representation that minimizes the area (or power) while satisfying the delay constraint.

The paper is organized as follows. The next section presents a brief overview of the SD number representation and the rules and conditions necessary to bound the length of carry propagation chains. Section III introduces the hybrid number representation and the operations performed in this number system. Section IV illustrates static CMOS implementations of various cells for the proposed representation. Section V analyses the area vs. delay and other tradeoffs associated with the choice of each of the possible representations. The framework permits a unified evaluation of the whole spectrum of two operand adders from full signed–digit adder to the ripple–carry adder. We then illustrate various possible implementations of adder trees for partial product accumulation in multiplication. Section VI presents conclusions and indicates possible future work. The appendix discusses the relationship between the GSD representation considered in [9] and the HSD number system proposed here.

II Signed–Digit Number Representation

For a given radix r, each digit x_i in an SD number system is typically in the range

$$-a \leq x_i \leq +a \quad \text{where} \quad \lceil \frac{r-1}{2} \rceil \leq a \leq r-1. \tag{1}$$

In such a system, a "carry–free" addition can be performed, where the term "carry–free" in this context means that the carry propagation is limited to a single digit position. In other words, the carry propagation length is *fixed* irrespective of the wordlength. The addition consists of two steps [10]. In the first step, an intermediate sum s_i and a carry c_i are generated, based on the operand digits x_i and y_i at each digit position i. This is done *in parallel* for all digit positions. In the second step, the summation $z_i = s_i + c_{i-1}$ is carried out to produce the final sum digit z_i. The important point is that it is always possible to select the intermediate sum s_i and carry c_{i-1} such that the summation in the second step does not generate a carry. Hence, the second step can also be executed *in parallel* for all the digit positions, yielding a fixed addition time, independent of the word length.

If the selected value of a in equation (1) satisfies the condition

$$\lceil \frac{r+1}{2} \rceil \leq a \leq r-1 \tag{2}$$

then the intermediate sum s_i and carry c_i depend only on the input operands in digit position i, i.e., on x_i and y_i. The rules for selecting the intermediate sum and carry are well known in this case [10]. The interim sum is $s_i = x_i + y_i - rc_i$ where

$$c_i = \begin{cases} 1 & \text{if} \quad (x_i + y_i) \geq a \\ -1 & \text{if} \quad (x_i + y_i) \leq -a \\ 0 & \text{if} \quad |x_i + y_i| < a \end{cases} \tag{3}$$

Note that for the most commonly used binary number system (radix $r = 2$), condition (2) cannot be satisfied. Carry–free addition according to the rules in (3) therefore cannot be performed with binary

operands. However, by examining the input operands in position $i-1$ together with the operands in digit position i, it is possible to select a carry c_i and an interim sum s_i such that the final summation $z_i = s_i + c_{i-1}$ never generates a carry. In other words, if one allows the carry c_i and interim sum s_i to depend on two digit positions, viz., i and $i-1$, then condition (2) can be relaxed and $a = \lceil \frac{r-1}{2} \rceil$ can also be used to accomplish carry–free addition as explained next.

Let x_i, y_i, x_{i-1} and y_{i-1} be the input digits at the ith and $(i-1)$th positions, respectively, and assume that the radix under consideration is $r = 2a$. This includes the case where $r = 2$ and $a = 1$. Let $\theta_i = x_i + y_i$ and $\theta_{i-1} = x_{i-1} + y_{i-1}$ denote the sums of the input digits at the two positions, respectively. Then, the rules for generating the intermediate sum s_i and carry c_i are summarized in Table 1. In the table, the symbol "$\times$" indicates a "don't care", i.e., the value of θ_{i-1} does not matter.

θ_i	$\theta_i = -2a$	$-2a < \theta_i < -a$	$\theta_i = -a$		$-a < \theta_i < a$	$\theta_i = a$		$a < \theta_i < 2a$	$\theta_i = 2a$
θ_{i-1}	$\times$	$\times$	$\theta_{i-1} \leq -a$	$\theta_{i-1} > -a$	$\times$	$\theta_{i-1} < a$	$\theta_{i-1} \geq a$	$\times$	$\times$
c_i	-1	-1	-1	0	0	0	1	1	1
s_i	0	$\theta_i + r$	a	$-a$	θ_i	a	$-a$	$\theta_i - r$	0

Table 1: Rules for selecting carry c_i and intermediate sum s_i based on $\theta_i = x_i + y_i$ and $\theta_{i-1} = x_{i-1} + y_{i-1}$ for radix $r = 2a$.

An equivalent form of this table for the special case when the radix r is equal to 2 was presented in [8].

III Hybrid Number Representation

Here, instead of insisting that every digit be a signed digit, we let some of the digits to be signed and leave the others unsigned. For example, every alternate or every third or fourth digit can be signed; all the remaining ones are unsigned. We refer to this representation as a Hybrid Signed-Digit (HSD) representation. In the following, we show that such a representation can limit the maximum length of carry propagation chains to any desired value. In particular, we prove that the maximum length of a carry propagation chain equals (d + 1), where d is the longest distance between neighboring signed digits.

It can be verified that addition in such a representation requires the carry in between all digit positions (signed or unsigned) to assume any value in the set $\{-1, 0, 1\}$ as in the SD system. Without loss of generality, assume that the radix is $r = 2$ and that every alternate digit is a signed digit, for the purpose of illustration. Then, the operations in a signed–digit position are exactly the same as those in the SD case. For instance, let x_i and y_i be radix–2 signed digits to be added at the ith digit position, and c_{i-1} be the carry into the ith digit position. Each of these variables can assume any of the three values $\{-1, 0, 1\}$. Hence $-3 \leq x_i + y_i + c_{i-1} \leq +3$. This sum can be represented in terms of a signed output z_i and a signed carry c_i as follows:

$$x_i + y_i + c_{i-1} = 2c_i + z_i \tag{4}$$

where $c_i, z_i \in \{-1, 0, 1\}$. In practice, the signed–digit output z_i is not produced directly. Instead, the

carry c_i and an intermediate sum s_i are produced in the first step, and the summation $z_i = s_i + c_{i-1}$ is carried out in the second.

The operations in an unsigned digit position are as follows. Let a_{i-1} and b_{i-1} be the bits to be added at the $(i-1)$th digit position; $a_{i-1}, b_{i-1} \in \{0,1\}$. The carry into the $(i-1)$th position is signed and can be $-1, 0$ or 1. The output digit e_{i-1} is restricted to be unsigned, i.e., $e_{i-1} \in \{0,1\}$. Hence the carry out of the $(i-1)$th position must be allowed to assume the value -1 as well. In particular

$$
\begin{array}{l}
\text{if} \quad (a_{i-1} = b_{i-1} = 0 \ \ \& \ \ c_{i-2} = -1) \ \ \text{then} \\
\quad c_{i-1} = -1 \ \ \text{and} \ \ e_{i-1} = 1 \\
\text{else} \\
\quad a_{i-1} + b_{i-1} + c_{i-2} = 2c_{i-1} + e_{i-1} \\
\qquad \text{where} \ \ c_{i-1}, e_{i-1} \geq 0 \\
\text{endif}
\end{array}
\tag{5}
$$

We next demonstrate that the carry propagates only between the signed digits. The addition consists of two steps:

Step 1 : The signed–digit positions generate a carry–out and an intermediate sum based only on the two input signed digits and the two bits at the neighboring lower order unsigned digit position. Let x_i and y_i be the signed digits to be added in the ith position and a_{i-1} and b_{i-1} be the unsigned digits (bits) in the $(i-1)$th position. The carry c_i and intermediate sum s_i at the ith (signed) digit position are selected based only on x_i, y_i, a_{i-1} and b_{i-1} according to Table 2.

$x_i + y_i$	$x_i\, y_i$	$a_{i-1}\ ,\ b_{i-1}$	c_{i-1}	s_i	c_i
-2	$\overline{1}\,\overline{1}$	$\times$	$\times$	0	-1
-1	$\overline{1}\,0$	$a_{i-1} = b_{i-1} = 0$	$\{-1,0\}$	$+1$	-1
-1	$0\,\overline{1}$	at least one of a_{i-1}, b_{i-1} is 1	$\{+1,0\}$	-1	0
0	$\overline{1}\,1$	$\times$	$\times$	0	0
0	$1\,\overline{1}$	$\times$	$\times$	0	0
0	0 0	$\times$	$\times$	0	0
$+1$	0 1	$a_{i-1} = b_{i-1} = 0$	$\{-1,0\}$	$+1$	0
$+1$	1 0	at least one of a_{i-1}, b_{i-1} is 1	$\{+1,0\}$	-1	$+1$
$+2$	1 1	$\times$	$\times$	0	$+1$

Table 2: Rules for selecting the carry c_i and intermediate sum s_i based on x_i, y_i, a_{i-1} and b_{i-1}, where x_i and y_i are signed digits, and $a_{i-1},\ b_{i-1}$ are unsigned digits.

In this table, $\overline{1}$ denotes -1 and $\times$ denotes a "don't care" as before. The first column of Table 2 indicates all possible values of the sum $(x_i + y_i)$. The second column indicates the individual digit values that lead to the sum in column 1. The third column indicates the possible values of a_{i-1} and

b_{i-1}. Together, these columns cover all possible inputs. The fourth column indicates the possible values of c_{i-1} which is the carry into the *i*th (signed) digit position. This carry into the signed digit position affects the carry out of the signed digit position (viz., c_i). Note that if $a_{i-1} = b_{i-1} = 0$ then c_{i-1} is *non–positive,* i.e., $c_{i-1} \in \{0, -1\}$. If at least one of a_{i-1} and b_{i-1} is 1, then c_{i-1} is *non–negative,* i.e., $c_{i-1} \in \{0, +1\}$. The polarity of c_{i-1} as defined by these mutually exclusive conditions (both a_{i-1}, b_{i-1} are zero and at least one of them is nonzero) is valid irrespective of what values x_i and y_i assume. The last two columns in Table 2 indicate the values of s_i and c_i, respectively, for each possible combination of x_i, y_i, a_{i-1} and b_{i-1}.

From the table, it is clear that the carry c_i out of the signed digit is independent of the carry into the previous unsigned $(i-1)$th digit position, viz., c_{i-2}. Hence, the carries out of, and the intermediate sums at *all* the signed digits positions can be calculated *in parallel* in the first step. Furthermore, from the table it is seen that whenever the carry c_{i-1} to be generated at the $(i-1)$th position is expected to be non–negative, s_i is selected to be non–positive and vice versa. In other words, s_i and c_{i-1} are guaranteed to have opposite polarity. Consequently, the addition $z_i = s_i + c_{i-1}$ can never generate a new carry. Thus, the carry propagation stops at the signed digit(s). The most important point is that it is possible to predict when c_{i-1} will be non–positive and when it will be non negative, just by looking at the operand digits a_{i-1} and b_{i-1}. It is not necessary to wait until the actual value of c_{i-1} becomes available; which makes it possible to break the carry propagation chain.

Step 2 : In the second step, the carries generated out of the signed digit positions ripple through the unsigned digits all the way up to the next higher order signed digit position, where the propagation stops as described above. The second step can also be carried out in parallel, i.e., all the (limited) carry propagation chains between the signed digit positions are executed simultaneously.

The most significant digit in any HSD representation must be a signed digit in order to incorporate enough negative numbers. All the other digits can be unsigned. For example, if the word length is 32 digits, then, the 32nd (i.e., the most significant) digit is a signed digit. The remaining digits are at the designer's disposal. If regularity is not necessary, one can make the 1st, 2nd, 4th, 8th and 16th (and 32nd) digits signed and let all the remaining digits be unsigned digits (bits). The addition time for such a representation is determined by the longest possible carry–propagation chain between consecutive signed digit positions (16 digit positions; from the 16th to the 32nd digit in this example).

The range of numbers covered by an *n* digit HSD representation depends on the number of signed digits and their positions. The SD representation (i.e., HSD with $d = 0$) has the highest range of $[-\frac{a(r^n-1)}{r-1}, +\frac{a(r^n-1)}{r-1}]$ for the digit set $[-a, +a]$. The largest positive number has the same value as above for other HSD representations. The smallest negative number, however, depends on d and the exact positions of the signed digits. Its value is obtained by setting all unsigned digits to 0 and all signed digits to $-a$. The smallest range corresponds to the HSD representation with $d = n-1$ (i.e., only the most significant digit is signed) and is equal to $[-ar^{n-1}, +\frac{a(r^n-1)}{r-1}]$. Thus, the range of an HSD representation with $d > 0$ includes fewer negative numbers than positive ones. The

range of an n digit HSD representation is in general larger than that of a conventional n digit radix complement representation. Therefore the conversion of an n digit HSD number to a conventional radix complement number requires $n+1$ digit positions. For the conversion, the algorithm presented in [11] can be used.

One extra signed digit position to the left of the most significant digit is sufficient to accommodate the result of the addition of two HSD format numbers. This, however, may lead to non uniform distance between signed digits. For example, when adding two binary HSD numbers with a uniform distance of 1 (alternate digits signed), one extra signed digit position is required to hold the result. This introduces non uniformity since the *two adjacent* most significant digits of the output are both signed. If the original format ($d = 1$) is to be preserved, then one must add *two* extra digits: one unsigned and one signed to accommodate the result of the addition. This is important in multi-operand addition (such as partial product accumulation in a multiply operation).

The HSD representation has another interesting property: there is no need to be restricted to a particular HSD format (with a certain value of d). The representation can be modified (i.e., the value of d can be changed) while performing addition (and consequently, other arithmetic operations) and this can be done in certain cases *without any additional time delay*. For instance, let x and y be two HSD operands, with uniform distances d_x and d_y, respectively, between their signed digits. Also assume that (d_y+1) is an integral multiple of (d_x+1) so that the signed digit positions of y are aligned with the signed digit positions of x (note that x has more signed digits than y under the stated assumption). Let $z = x+y$ be their sum, having a uniform distance d_z between its signed digits. The possible values of d_z which are interesting from a practical point of view are $0, d_x$ and d_y. If we set $d_z = d_y$ then the above addition will take exactly the same time as an addition of two HSD operands with uniform distance d_y producing an HSD result with distance d_y. Setting $d_z = d_x$ (and clearly, $d_z = 0$) will reduce the addition time even further, since the introduction of extra signed digits results in shorter carry propagation chains. For example, suppose that $d_x = 0$ (all digits are signed) and $d_y = 1$ (alternate digits are signed). If d_z equals 1, then the delay required to perform the addition is the same as that required to add two HSD numbers with the same distance $d = 1$ to generate an output with $d_z = 1$. This format conversion flexibility (without any additional time delay penalty) can be useful, as illustrated later in the discussion of partial product accumulation.

The HSD representation is related to the GSD (Generalized Signed Digit) representation considered in [9]. In the special case when the distance between adjacent signed digits is uniform, our HSD representation becomes identical to the GSD representation [12]. Adopting the algorithm summarized in Table 2 can be considered as an efficient implementation of GSD addition in this special case. When the distance between signed digits is non–uniform, however, HSD representation is no longer equivalent to the GSD scheme. Such a non–uniform distance between signed digits, corresponds (in some sense) to using different radii for different digit positions; a concept that is clearly beyond the scope of the GSD framework. It should be noted that the proposed HSD representation and the algorithm in Table 2 is also valid and yields equally efficient implementation even when the distance d between signed digits is non-uniform. The appendix discusses the relationship between the HSD and

GSD schemes in further detail.

IV Static CMOS Implementation

We now present an implementation of the signed and unsigned digit adder cells. Kuninobu et. al. [5] have proposed some of the most efficient designs for a cell that adds radix–2 signed–digit operands. We have therefore adopted the same design methodology.

Addition of bits at the unsigned digit position : Inputs to the cell are the bits a_{i-1}, b_{i-1} and the incoming carry c_{i-2}. The outputs are the carry out c_{i-1} and the final sum e_{i-1}. The output digit e_{i-1} is unsigned and can take one of the two values 0 or 1. The carries c_{i-2} and c_{i-1} can take values in $\{-1, 0, 1\}$ and need two bits for encoding. Following [5] we encode each of these carries using two unipolar binary variables, each requiring a single bit.

From Table 2 note that

(i) When both a_{i-1} and b_{i-1} are 0, the carry c_{i-1} is non–positive (i.e., $c_{i-1} \in \{-1, 0\}$) and the intermediate sum s_i is non–negative (i.e., $s_i \in \{0, 1\}$).

(ii) When at least one of a_{i-1} and b_{i-1} is nonzero, the carry c_{i-1} is non–negative (i.e., $c_{i-1} \in \{0, 1\}$) and the intermediate sum s_i is non–positive (i.e., $s_i \in \{-1, 0\}$).

Let $w_{i-1} = 1$ if condition (i) is satisfied (i.e., $a_{i-1} = b_{i-1} = 0$) and $w_{i-1} = 0$ if condition (ii) is satisfied (i.e., at least one of a_{i-1}, b_{i-1} is 1). Then, the variable v_{i-1} defined by

$$v_{i-1} = w_{i-1} + c_{i-1} \tag{6}$$

is always non–negative. In effect, the carry c_{i-1} is expressed as the difference of two bits v_{i-1} and w_{i-1}, i.e., $c_{i-1} = v_{i-1} - w_{i-1}$. Similarly, the incoming carry c_{i-2} is also expressed as a difference of two bits

$$c_{i-2} = v_{i-2} - w_{i-2} \tag{7}$$

where $v_{i-2}, w_{i-2} \in \{0, 1\}$. Thus, the cell in the $(i-1)$th (unsigned digit) position should generate w_{i-1}, v_{i-1} and e_{i-1} from $a_{i-1}, b_{i-1}, v_{i-2}$ and w_{i-2}. Note that all of these variables are binary and can be encoded by a single bit. The above equations are *algebraic*, but the symbols used in these equations (i.e., w_{i-1}, v_{i-1} etc.) can also be used to indicate the corresponding *logical* variables. The distinction follows from the context.

From the above, the following logical equations are obtained [13] :

$$w_{i-1} = \overline{a_{i-1} + b_{i-1}} \tag{8}$$

$$v_{i-1} = v_{i-2}\overline{w_{i-2}} + (\overline{a_{i-1} \oplus b_{i-1}})(v_{i-2} + \overline{w_{i-2}}) \tag{9}$$

$$e_{i-1} = (a_{i-1} \oplus b_{i-1}) \oplus (v_{i-2} \oplus w_{i-2}) \tag{10}$$

The logic diagram of the cell consisting of 32 transistors is shown in Figure 1(a). Here, it is assumed that the XOR (or XNOR) can be implemented with pass gates with 6 transistors when both the inputs are available only in true (uncomplemented) form, and with 4 transistors if one of the inputs is

available in true as well as complemented form [14]. Note that the cell actually generates $\overline{v_{i-1}}$. This is done to reduce the number of transistors and the delay associated with the critical path.

The addition of signed digits : The inputs to this cell are the signed digits x_i, y_i and the carry signals w_{i-1} and v_{i-1}. A signed digit x is encoded in two bits $x^s x^a$, with, -1, 0 and 1 encoded by 11, 00, and 01, respectively. The outputs of the cell are the carry signals v_i , w_i and the bits that represent the output signed digit (z_i^s , z_i^a). In order to reduce the critical path delay and the transistor count, the cell accepts and produces sign bits of signed digits in complemented form (i.e., $\overline{x_i^s}$, $\overline{y_i^s}$ and $\overline{z_i^s}$).

Once again, from Table 2, note that

(iii) When at least one of x_i and y_i is negative, c_i is non–positive.

(iv) When both x_i and y_i are non–negative, c_i is non–negative.

Let $w_i = 1$ when condition (iii) is satisfied and $w_i = 0$ when condition (iv) is satisfied. Then, the variable v_i defined by

$$v_i = w_i + c_i \tag{11}$$

is always non–negative. Thus, the carry c_i is represented by the difference of two bits v_i and w_i.

From Table 2 and conditions (i) and (ii), it can be seen that the variable q_i defined by

$$q_i = w_{i-1} - s_i \tag{12}$$

is always non–negative as well. The output signed digit z_i satisfies

$$\begin{aligned} z_i &= s_i + c_{i-1} && (13)\\ &= (w_{i-1} - q_i) + (v_{i-1} - w_{i-1}) \\ &= v_{i-1} - q_i && (14) \end{aligned}$$

where the substitutions are from (6) and (12). The signed digit cell therefore generates the output bits w_i, v_i and the intermediate bit q_i from x_i, y_i and w_{i-1}. It then generates the final sum z_i from q_i and v_{i-1} according to equation (14). From the above, one obtains the following logical equations.

$$\begin{aligned} w_i &= x_i^s + y_i^s = \overline{\overline{x_i^s} \cdot \overline{y_i^s}} && (15)\\ s_i^a &= x_i^a \oplus y_i^a && (16)\\ q_i &= s_i^a \oplus w_{i-1} && (17)\\ \overline{z_i^s} &= \overline{q_i \cdot \overline{v_{i-1}}} && (18)\\ z_i^a &= \overline{q_i \oplus \overline{v_{i-1}}} && (19)\\ \overline{v_i} &= s_i^a \cdot w_{i-1} + x_i^s \cdot y_i^s + \overline{x_i^a} \cdot \overline{y_i^a} && (20) \end{aligned}$$

Details of the derivation can be found in [13]. The logic diagram of the cell, shown in Figure 1(b), is very similar to that of the cell presented in [5]. The only difference is the polarity of the input

and output variables. The design in [5] accepts and produces all variables in true (uncomplemented) form. The cell shown in Figure 1(b) and the SD adder cell presented in [5] both require 42 transistors. However, in the HSD representation, where alternate digits are unsigned, the cell in every alternate digit position is that of Figure 1(a), which requires only 32 transistors. Thus, a group of two adjacent digit positions requires 74 transistors in our design, instead of 84 transistors that are needed for two of the cells in [5]. The savings in the number of transistors is even higher if an HSD representation with $d \geq 2$ is employed.

Following [5], for the sake of consistent comparisons with the designs presented therein, we also make the (somewhat crude) assumption that the delay associated with XOR, XNOR, A–O–I (AND–OR–INVERT), O–A–I (OR–AND–INVERT) gates is 1.5 units, while all other delays are 1 unit. Similar assumptions were used in [15] to estimate the total delays in the units of an inverter or basic two input NAND/NOR gate delay. Then, the critical path delay of our design is 6 units while that of the SD-based design in [5] is 5 units. We would like to point out that the cells shown in Figure 1 were designed to minimize critical path delay and transistors for the case where alternate digits are signed. In this implementation, the *longest* path begins at the inputs of the (lower order) unsigned digit (such as the $(i-1)$th digit). It traverses digit positions i (signed) and $i+1$ (unsigned) via signals w_{i-1}, v_i and v_{i+1} and terminates at the signed digit position $i+2$. The delay associated with this path is only 5.5 units. Note that the signed digit input signals x_i^a and y_i^a at the ith digit position also propagate up to the $(i+2)$th digit output through four complex gates (via signals s_i^a, v_i and v_{i+1}). The delay associated with this path is 6 units and it is the critical path. It is interesting to note that the path which spans the maximum number of digit positions is not critical. Another path between consecutive signed digits has maximum delay and hence is the critical path, even though its span is smaller by one digit position.

Note that the cell in Figure 1(a) generates $\overline{v_k}$ (where $k = i-1$). If there are more than one unsigned digits between two signed digits (say, for instance, that there are two bits between two signed digits), the next cell will take as input the complemented form of v_k, i.e., $u_k = \overline{v_k}$. In terms of u_k and w_k, the carry into the $(k+1)$th position is expressed as $c_k = v_k - w_k = 1 - u_k - w_k$. The logic diagram of a cell that accepts this input is shown in Figure 2. At its output, this cell again *inverts* the v_{k+1} signal (with respect to the polarity of v_k), i.e., it generates v_{k+1} in true form. It is possible to add an inverter to the cell in Figure 1(a) and make v_k available in true form. This would obviate the need to have a different type of cell (viz., the cell in Figure 2) and only the cell in Figure 1(a) with an additional inverter would be sufficient. This inverter, however, falls on the critical path and this way, the number of (extra) inverters equals the number of unsigned digits traversed minus 1. With the introduction of the third cell, there is at most one extra inverter when the number of unsigned digits between two signed digits is *even*. If d is the number of unsigned digits between two neighboring signed digits, then the critical path delay is

$$\tau_{critical} = \begin{cases} (1.5+1.5)+1.5d+[1.5] \\ \quad = 4.5+1.5d \quad \text{if } d \text{ is odd} \\ (1.5+1.5)+1.5d+1+[1.5] \\ \quad = 5.5+1.5d \quad \text{if } d \text{ is even} \end{cases} \tag{21}$$

Here, the two delays of 1.5 units in parenthesis are due to the two complex gates in the lower order signed digit cell. The last 1.5 units of delay (shown within the square brackets) is associated with the XNOR gate at the higher order signed digit where the carry propagation terminates. The terms in between are proportional to d since the carry ripples through all the unsigned digit positions. The transistor count can be similarly calculated. Assume that there are exactly g groups of $d+1$ digits each, where the most significant digit in each group is signed and all other digits are unsigned. In other words, the word length n satisfies $n = g \times (d+1)$. The transistor count for a group and a complete adder of this type are denoted by N_{group} and N_{total}, respectively, and are given by

$$\begin{aligned} N_{group} &= \begin{cases} 32d+42 & \text{if } d \text{ is odd} \\ 32d+2+42 & \text{if } d \text{ is even} \end{cases} \\ N_{total} &= g \times N_{group} = \frac{n \cdot N_{group}}{d+1} \\ &= \begin{cases} n(32+\frac{10}{d+1}) & \text{if } d \text{ is odd} \\ \\ n(32+\frac{12}{d+1}) & \text{if } d \text{ is even} \end{cases} \end{aligned} \tag{22}$$

Next, we analyze the cost and performance of some basic arithmetic operations when implemented in the HSD number system.

V Cost and Performance Tradeoffs for HSD Implementations

Based on the above discussion, it is seen that the HSD representation is very flexible and offers a wide variety of choices to the designer. Increasing d trades off higher delay for lesser area. In this section we first evaluate the whole spectrum of adders: from ripple–carry to full signed–digit adders under the unifying HSD framework. Next, we illustrate different implementations of adder trees for partial product accumulation in a multiply operation.

Two operand addition : The first operation we consider is the addition of two operands. Equations (21) and (22) in the previous section evaluate the critical path delay and transistor count as functions of the distance d between signed digits. This allows us to evaluate the whole spectrum of adders from ordinary ripple–carry adders at one end to the (fully) SD adders on the other extreme. The plots for word length $n = 24$ are shown in Figures 3(a) to 3(d). The word length was chosen to be 24 since this is the number of significand bits in the IEEE standard for single precision floating–point numbers. In these figures, the measures of interest (area, delay, area $\times$ delay etc.) are plotted as a function of d, the

distance between adjacent signed digits. The point $d = 0$ corresponds to the SD representation, where every digit is signed. The transistor count and delay for this case are based on the design presented in [5]. The point $d = 23$ at the other extreme corresponds to the ripple–carry adder with all digits unsigned. Here, the cells are 1 bit full adders with a transistor count of 22 and critical path delay of 1.5 per cell. The corresponding values for a carry–look–ahead adder are also shown in each of these figures as a separate point (denoted by the symbol " ⊛") with abscissa $d = 23$. Here the blocking factor (or the fan–in) of the carry–look–ahead tree was assumed to be 4.

Figure 3(a) shows transistor count vs. the distance between signed digits (d). Figure 3(b) shows critical path delay vs. d; Figure 3(c) shows (area $\times$ time delay) vs. d and Figure 3(d) shows (area$^2\times$ delay) vs. d. Note that (area$^2\times$ time delay) can be considered to be a rough estimate of (area $\times$ power $\times$ time delay) because the power consumed is proportional to $CV^2 f$, where C is the overall capacitance, V is the supply voltage and f is the effective frequency (i.e., f includes the effect of both the clock rate and the actual transition or switching rate). V and the clock rate are fixed parameters. Assuming that f depends only on the clock rate, it too is a constant. The power consumed is therefore approximately proportional to the overall capacitance C, which is in turn proportional to the transistor count or area.

These plots do not take into account the overhead of converting the SD (or HSD) sum to two's complement. For a two–operand addition, the cost of converting the redundant output back to two's complement could make the redundant adders slower and/or bigger than the conventional carry–look–ahead or carry–skip adders. In more complex applications involving multi–operand addition, however, the conversion to two's complement is needed only once at the end. If the number of operands to be added is large, the time per addition is an important factor. The performance numbers depicted in the figures are more meaningful in this context rather than a simple two operand addition. Also, the transistor count alone is not a sufficiently accurate estimate of the area, since multi–operand addition usually involves trees which need a significant amount of routing area. It is a fairly good estimate of the complexity nonetheless.

From Figure 3(c), it is seen that the (fully) signed–digit implementation [5] is AT optimal. It requires maximum area but takes minimum time. A ripple–carry adder, on the other hand takes maximum time and minimum area. The alternate–signed–digit ($d = 1$) adder illustrated above reduces the transistor count from 84 to 74 (for two digit positions), but increases the critical path delay from 5 to 6 units. It closely matches the AT performance of the SD adder. Other bounded–carry–propagation adders take less area and more time, but the AT product keeps on increasing with d. The carry–look–ahead adder also has a very good AT product, which is fairly close to the optimal value. The last two points (d = 22 and 23) on the A vs. d curve in Figure 3(a) illustrate that there is a sizable increase in the area when one introduces even a single signed digit. This is due to the fact that the unsigned digit cell must now handle a carry in the set $\{-1, 0, 1\}$ which makes it more complex than the full adder cells of a ripple–carry adder. In Figure 3(a), note that at every point between $d = 12$ and $d = 23$, the number of signed digits is 2, since the most significant or 24th digit is signed and there is one

additional signed digit in the word. As d is increased from 12 to 22, the position of the signed digit shifts from the 13th to the 23rd place. Thus the distance between signed digits is non–uniform when $12 < d \leq 22$. However, the total number of signed digits and hence the total number of unsigned digits, remain 2 and 22, respectively, for d in this range. Hence, the area is constant for all d values in the range from 13 to 22. The critical path, on the other hand, increases linearly with the longest distance between signed digits as illustrated in Figure 3(b). Also, beyond $d > 20$, the carry chain between signed digits is long enough to render the total delay (i.e., the propagation delay through the chain plus the complex gates at the ends) of the HSD adder *higher* than that of the ordinary ripple–carry adder. Figures 3(c) and 3(d) show that increasing d beyond a certain point makes AT and A^2T for HSD adders worse than that for ordinary ripple–carry adders. Finally from Figure 3(d), note that the HSD adder with $d = 1$ is A^2T optimal.

The continuum of choices available can be exploited to obtain the most suitable implementation under area or delay constraints. For instance, if the clock period is estimated to be about 10 gate delays, then from Figure 3(b), it is seen that all implementations with $d \leq 3$ satisfy the delay constraint. If the area is to be minimized, then $d = 3$ is the appropriate choice, i.e., every 4th digit should be signed. Thus, one can select the minimum area solution that meets the given delay constraint. If minimization of area is less critical, then one can select the full SD representation that yields the fastest execution time. Similarly, given a constraint on area, from Figure 3(a), one can obtain d values that satisfy the area constraint. Then from Figure 3(b), the designer can select that value of d which minimizes the execution delay, while satisfying the area constraint.

Multi–operand addition (Partial product accumulation) : A 64×64 multiplier with radix–4 modified Booth recoding [10] generates 32 partial products, that can be accumulated with a redundant adder tree having 5 levels. The final conversion from a redundant to two's complement format is carried out by a carry–look–ahead scheme. Redundant binary adder trees have proved to be highly efficient in accumulating partial products in fast, parallel, tree–based multiplication schemes. For instance, the 64×64 multiplier reported in [5] is faster, smaller (transistor count) and easier to lay out than a Wallace-tree-based multiplier. In the redundant adder tree, the reduction in the number of operands is from 2 to 1, while in a tree of full adders (or (3,2) counters), the reduction ratio is 3 to 2. Also, the trees based on full adders have diagonal connections for carries, which makes their layout considerably more difficult. Here, we illustrate a few HSD based implementations of redundant adder trees and the tradeoffs associated with each implementation.

We begin with the fully signed–digit format. An efficient design of an SD adder tree was presented in [5]. At the top level of the tree, the partial products generated are in two's complement format. Hence, one need not use the complex cells that add two SD numbers. In fact the cell at the top level can be considerably simplified as was shown in [5]. Let A and B be two operands (partial products) in two's complement format. Then the addition operation $A + B$ can be rewritten as $A - (-B)$. The generation of $-B$ involves complementing every bit of B and adding a "1" in the least significant digit position. Complementing all the bits of B is easy and can be done in parallel. The addition of 1 in the least significant digit position is postponed to the next level of the adder tree. One is then left with

the subtract operation $A - \tilde{B}$ at the top level of the tree, where $\tilde{B} + 1 = -B$. Generation of a signed digit output by subtraction of two operands in two's complement format is very easy. Note that the only possible outputs of the bit–wise subtraction can be $\{-1, 0, 1\}$, each of which is a valid signed digit output. Thus, there is no carry propagation whatsoever between any digit positions. Such a digit wise subtraction can be implemented very economically by employing the cell shown in Figure 4 [5]. The cell accepts two bits *a* and *b*, each in $\{0,1\}$, and generates a signed digit output resulting from the operation $a - b$. As seen in the figure, there is no carry propagation and the digit positions are independent of each other. The critical path delay is 1.5 units and two digit positions need 20 transistors (10 per digit position).

All the cells at the remaining four levels are identical to the full SD adder cell presented in [5], requiring 42 transistors and having a critical path delay of 5 units. The transistor count of the SD adder tree is estimated to be 67 K, including the final conversion stage. If the final carry–look–ahead conversion from redundant to two's complement format handles groups of 4 digits in parallel, there is a $\lceil \log_4 128 \rceil = 4$-level look–ahead tree. Then, the delay for the final conversion is approximately 20 units. Assuming that the delay required for the generation of partial products is 5 units, the total delay for the SD tree-based multiplier is 46.5 units (5 + 1.5 + 4 $\times$ 5 + 20).

Next, we consider an adder tree based on the HSD representation with $d = 1$, (i.e., every alternate digit is signed). This representation can be expected to make the redundant binary adder tree even smaller and easier to lay out for two reasons. First, the number of wires to be routed is only $\frac{3}{4}$ of that in the SD scheme, since every alternate digit is unsigned and needs only 1 bit for encoding. Second, the cells in the alternate digit position of an HSD adder handle unsigned bits and have a smaller transistor count. Once again, the cells at the top level of the tree can be considerably simplified since all the operands are in two's complement format. The simplified cells are shown in Figures 5(a) and 5(b). In Figure 5, the cell in the signed digit position has 20 transistors, while the one in the unsigned digit position has 24. The carries at all digit positions can always be selected to be non negative and hence need only one bit to encode. The operation of the cells is as follows. Let *A* and *B* be the two operands in two's complement format. The cells accept bit–wise complemented inputs $\overline{a_i}$ and $\overline{b_i}$ and generate the sum $(A + B)$ in HSD format where every alternate digit is signed. The inputs are complemented in order to reduce the critical path delay without increasing the transistor count. Moreover, the cells generate the sign bit of the signed digit output in the complemented form, as required by the next level. The transistor count for two digit positions (signed and unsigned) is 20 + 24 = 44. The length of the carry propagation chain is smaller than the case when both operands are signed digits and the critical path delay is 4 units.

The overall transistor count estimate for this tree (including the final conversion stage) is 72 K. The total delay for this HSD tree-based multiplier is (5 + 4 + 4 $\times$ 6 + 20 =) 53 units. Note that the SD tree has a smaller overall transistor count. The reason for this is as follows. In a tree, the number of leaf nodes = (1 + number of internal nodes). Hence, the number of adders at the top level equals (1 + number of adders at all other levels of the tree). At the top level, the SD-based scheme is more

economical as demonstrated by the cells in Figures 4 and 5. Hence, the overall transistor count for the SD tree is smaller then that for an HSD tree with $d = 1$. While these estimates make it appear that the HSD-based tree has no advantage, we next show that the format conversion flexibility of the HSD system can be utilized to significantly reduce the transistor count (to below that required by the SD tree) without increasing the critical path delay.

As mentioned earlier, the HSD representation allows format conversion during the addition of two operands without any extra delay penalty. This suggests that one can use a combination of HSD variants to obtain implementations that would require fewer transistors and lesser routing. We illustrate this with a tree that utilizes both SD (i.e., HSD with $d = 0$) and HSD with $d = 1$ number representations. At the top level of the tree where the inputs (partial products) are in two's complement format, it is more economical (in terms of the number of transistors used) to use the cell shown in Figure 4, that produces an output in the full SD format. At the subsequent levels, the HSD (with $d = 1$) cells are more economical. To exploit the advantages of both, it would be best to generate full SD representation at the top level and then switch to the HSD representation at the next level. Unfortunately, if both operands in the addition have the full SD format then the only output format possible is full SD.

If however, one of the operands is in SD format and the other is in HSD format with $d = 1$, then it is possible to produce an HSD format output (with $d = 1$) in the same time delay that is required to add two HSD format operands with $d = 1$ as explained next. Let the unsigned digit position under consideration be the $(i-1)$th, and the signed and unsigned digits and the carry to be summed be denoted by z_{i-1}, d_{i-1} and c_{i-2}, respectively. Then $-2 \leq \theta_{i-1} = z_{i-1} + d_{i-1} + c_{i-2} \leq +3$. Let the output bit (unsigned digit) be e_{i-1}, and the carry generated at the $(i-1)$th position be c_{i-1}. Then, for each possible value of θ_{i-1}, the (unique) values of c_{i-1} and e_{i-1} are determined from the relation

$$\theta_{i-1} = 2c_{i-1} + e_{i-1} \tag{23}$$

where $e_{i-1} \in \{0,1\}, c_{i-1} \in \{-1,0,1\}$. The ith output digit is signed and can therefore stop the carry propagation chain. In order to do so, all it needs is the information about the polarity (non positive or non negative) of the carry in, viz. c_{i-1}. The polarity of c_{i-1} can be determined just by looking at the operands z_{i-1} and d_{i-1}, without knowing the actual value of c_{i-1}. It is easy to verify that c_{i-1} is non–negative when [($z_{i-1} = +1$) or ($z_{i-1} = 0$ and d_{i-1} = 1)] and non–positive in all other cases.

Thus, in a design that exploits both SD and HSD (with $d = 1$) formats, half the cells at the top level are of the type shown in Figure 4 and generate partial sums in full SD format. The remaining partial products are summed using the cells shown in Figure 5 to generate partial sums in the HSD format. At the next level of the tree, the corresponding pairs of partial sums (one in SD format and one in HSD format) are then added together as explained above, to generate the output in the HSD format. From this point on, all the operands are in the HSD format throughout all the remaining levels of the tree. The HSD format final sum is then converted back to two's complement format as explained above. Henceforth, this scheme is referred to by the acronym MHSSD (Mixed Hybrid Signed and Signed Digit) implementation. Note that alternate digit positions in MHSSD have both digits signed. Hence

the cell shown in Figure 1(b) is sufficient. In order to generate c_{i-1} and e_{i-1} according to equation (23), only one other cell that accepts a signed digit, an unsigned digit (bit) and a signed carry, and generates an unsigned output bit and a signed carry needs to be designed. The design methodology is identical to that used for the cells shown in Figure 1 and is omitted for the sake of brevity (the details can be found in [13]). The resulting cell (that occupies alternate digit positions) has 34 transistors instead of 32 that a cell in Figure 1(a) needs. This is a very small increase which is far more than offset by the savings at the top level. The net result is a considerable reduction in the overall transistor count: with this scheme, the overall transistor count is approximately 66 K, which is smaller than the transistor count of both SD or HSD trees. The critical path delay for level 2 (the one that adds one SD and one HSD operand to produce one HSD operand) is still 6 units. Consequently, the overall delay of this scheme is the same as that of the HSD scheme with $d = 1$, i.e., 53 units.

It should be emphasized that transistor count alone is by no means an accurate measure of the area. It is well known that transistors are cheap to implement but communication, especially non–local interconnections are expensive in VLSI. Hence, there is a great deal of emphasis on minimizing the interconnections and trying to keep them local. In a tree architecture, the interconnections are inherently non-local. The MHSSD tree proposed here has only $\frac{3}{4}$th the number of wires to be routed at each level (except the top level) as compared to an SD tree. Hence the routing area for the MHSSD tree is expected to be significantly smaller than that of the SD tree. The transistor count of the MHSSD tree is smaller than that of the SD tree as mentioned above. Hence, the total area required by the MHSSD tree can be expected to be smaller than that of SD tree.

From the delays above, we have $\frac{(\text{delay})_{MHSSD}}{(\text{delay})_{SD}} = 1.1398$ and therefore, an overall area ratio of $\frac{(\text{Area})_{MHSSD}}{(\text{Area})_{SD}} \leq 0.8774$ will make MHSSD tree AT–optimal (among all the schemes that utilize signed digits). Similarly an area ratio of 0.937 or less will make the MHSSD scheme A^2T–optimal. As mentioned above, A^2T can be thought to be a crude approximation of the power consumed. Thus the MHSSD scheme is expected to result in an implementation that consumes less power. This can be a significant advantage if power minimization is an objective.

Other combinations of two or more HSD variants are possible and may lead to further reduction in area. Depending on the objective function or the constraints at hand, the designer can select the particular scheme that yields the most suitable implementation.

VI Conclusion

A novel, hybrid number representation has been proposed and was shown to lead to a bounded carry propagation during addition. The system uses a mixture of unsigned and signed digits to represent a number. It was demonstrated that the maximum length of a carry propagation chain in such a system is limited to the (longest) distance between adjacent signed digits and can therefore be set to any desired value from 1 to the entire word length by selecting the position(s) of the signed digits. This reveals a continuum of number representations from two's complement on one hand to the completely signed–digit system on the other. This framework was used to analyze the area and time

delay tradeoffs associated with each representation. It also permits a unified performance analysis of the whole spectrum of adders based on these number systems.

Implementations based on the HSD representations were shown to yield fast and compact adder and multiplier realizations. The format conversion flexibility of HSD representation opens up new possibilities of combining two or more HSD variants in order to obtain more suitable implementations. This was illustrated with a multiplier design that exploits the advantages of both the HSD with $d = 1$ and $d = 0$ representations. Besides demonstrating the flexibility offered by the above theoretical framework, the design has several attractive performance attributes: it requires a small area (smaller than multipliers that use only a single representation in the redundant adder tree), and is likely to consume low power.

Other arithmetic operations such as division, square root extraction and elementary function evaluation have been accelerated by using SD representation. For these operations, the HSD framework would provide a continuum of choices between ordinary and full signed digit implementations that trade off area for speed.

VII Appendix

Relation between HSD and GSD representations : In [9] Parhami presented a unified treatment of several signed–digit schemes under a general framework called the GSD (Generalized Signed Digit) number representation. In the GSD formulation, each digit in a radix r positional number system can take any value in the interval $[-\alpha, +\beta]$. Conditions on r, α and β that are necessary in order to perform carry–free addition were presented, and equations to perform the carry–free addition were derived in [9]. Besides the radix–r SD representation, this formulation also includes stored–borrow/stored–carry type representations as special cases. However, all the digit positions in GSD are uniform, i.e., the range of values a digit can assume and the way the digit-wise operations are carried out is *the same* for all digit positions. Our representation, on the other hand, deliberately introduces non uniformity in the digit positions in order to reduce the transistor count (and area). Thus, some digits are allowed to be signed and others are left unsigned which makes the range of values a digit can assume, non uniform. Also, the operations performed at signed and unsigned digit positions are quite different as illustrated above.

In the special case when the number of unsigned digits between any two adjacent signed digits is the same, say d, our HSD representation can be considered to be a special case of the GSD representation with radix r^{d+1} [12]. In a group of $d+1$ digits (d unsigned and 1 signed), if the signed digit assumes the least significant position then the limits of the interval of allowed digit values are given by $-\alpha = -(r-1)$ and $\beta = r^{d+1} - 1$, respectively. If the signed digit assumes the most significant position, the corresponding limits are $-\alpha = -(r-1) \cdot r^d$ and $\beta = r^{d+1} - 1$. Obviously, the second choice is better, since it allows more values to be represented with the same number of bits. For example, consider the HSD representation with $r = 2$ and $d = 1$ (i.e., every alternate digit is signed). Here, a group of two digits, viz, a signed digit and the lower order adjacent unsigned bit can be interpreted as a radix–4 hybrid signed digit which can take any value in the range $[-\alpha, +\beta]$, where $\alpha = 2$ and

$\beta = 3$. Note that this range is asymmetric and different from the range $[-3,+3]$ that a conventional radix–4 signed–digit system uses (see equation (1)). In particular, the number of values a digit can assume is smaller, even though the same number of bits (3 bits) is used to represent each digit.

When the distance between the signed digits is non-uniform, however, the HSD representation is no longer a special case of the GSD representation. Such a non–uniform distance between signed digits, corresponds (in some sense) to using different radii for different digit positions; a concept that is clearly beyond the scope of the GSD framework. A non–uniform distance between signed digits along with the format conversion property of the HSD representation could be useful in multi-operand addition. For instance, consider $r = 2$ and $d = 1$. In order to prevent overflow (in a two operand addition), one can add an extra signed digit to the most significant position, leading to a non uniform d. If the result is then added to another HSD number with $d = 1$, the sum output can be rendered in the uniform $d = 1$ format. Another possible use of non uniform d is in case when the word length is a prime number such as 53, which is the total number of significant bits (a 52 bit mantissa + 1 hidden bit) in the double precision floating–point format prescribed in the IEEE standard 754.

Moreover, even in the case when the distance between the signed digits is uniform, a brute force implementation of the carry–free addition according to equation (3) in Section II will lead to far more complex logic, larger area and longer critical path delay. For instance, consider the case when the signed digit occupies the most significant position in a group of $d+1$ adjacent digits $i,\ i-1;\cdots,(i-d+1)$. If this group is interpreted as a signed digit with radix 2^{d+1}, then the carry out of the signed digit position c_i depends on the *values* of this and the previous (radix 2^{d+1}) digits. Note that in order to restrict the carry propagation to a single digit position, equation (2) must be satisfied. This equation implies that the digits must be allowed to take any value in the interval $[-(2^{d+1}+1),+(2^{d+1}+1)]$. The smallest value the radix–2^d digit can take is $-\alpha = -2^d$, which does not meet this condition. Hence, only the scheme where the carry depends on two digit positions is feasible. In other words, the carry out c_i depends on all of the radix–2 operand digits in positions $i-1,\cdots,i-d+1$ *as well as those in digit positions* $i-d,\cdots,\ i-2d+1$ or a total of $4(d+1)$ binary digits (or $4d+6$ bits, since there are exactly two signed digits, each requiring one extra bit). The carry generation logic in such a case would be prohibitive. In contrast, adopting our representation and the use of Table 2 shows that the carry out c_i depends only on four binary operand digits, viz., those in digit positions i and $i-1$ or equivalently, only on six bits. Thus, using the proposed HSD representation is more efficient (in terms of transistor count, as well as delay) than using the conventional higher radix signed–digit representation.

The GSD summation algorithm in [9] requires a comparison of the sum $x_i + y_i$ with certain constants to determine the carry. A brute force implementation might need the actual value of the sum $x_i + y_i$. However, with proper selection of constants and digit encodings as described in [9], only a few bits of each operand may be actually needed to determine the outcome of the comparison, thereby obviating the need to actually evaluate the sum. Our HSD representation utilizes only a few bits (3 bits) of each operand as illustrated above and can be considered to be an efficient implementation of

GSD addition in the special case when the distance between adjacent signed digits is uniform.

References

[1] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 389–400, Sep. 1961.

[2] M. D. Ercegovac and T. Lang , "Fast multiplication without carry-propagate addition," *IEEE Transactions on Computers*, vol. C-39, pp. 1385–1390, Nov. 1990.

[3] M. D. Ercegovac and T. Lang, "Redundant and on-line CORDIC: application to matrix triangularization and SVD," *IEEE Transactions on Computers*, vol. C-39, pp. 725–740, Jun. 1990.

[4] M. J. Irwin and R. M. Owens, "Digit–pipelined arithmetic as illustrated by the paste–up system: A tutorial ," *IEEE Computer*, pp. 61–73, Apr. 1987.

[5] S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, and N. Takagi, "Design of high speed MOS multiplier and divider using redundant binary representation," in *Proc. of the 8th Symposium on Computer Arithmetic*, pp. 80–86, 1987.

[6] H. Makino, Y. Nakase and H. Shinohara, "A 8.8–ns 54 $\times$ 54–bit multiplier using new redundant binary architecture," in *Proceedings of the International Conference Computer Deign (ICCD), Cambridge, Massachusetts*, pp. 202–205, Oct. 1993.

[7] H. R. Srinivas and K. K. Parhi, "A fast VLSI adder architecture," *IEEE Journal of Solid-State Circuits*, vol. SC-27, pp. 761–767, May 1992.

[8] N. Takagi, H. Yasuura, and S. Yajima, "High-speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Transactions on Computers*, vol. C-34, pp. 789–796, Sep. 1985.

[9] B. Parhami, "Generalized signed-digit number systems: a unifying framework for redundant number representations," *IEEE Transactions on Computers*, vol. C-39, pp. 89–98, Jan. 1990.

[10] I. Koren, *Computer Arithmetic Algorithms*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1993.

[11] S. M. Yen, C. S. Laih, C. H. Chen and J. Y. Lee, "An efficient redundant–binary number to binary number converter," *IEEE Journal of Solid State Circuits*, vol. SC-27, pp. 109–112, Jan. 1992.

[12] B. Parhami., Personal Communication.

[13] D. S. Phatak and I. Koren., "Hybrid Signed–Digit number systems: A unified framework for redundant number representations with bounded carry propagation chains," Tech. Rep. TR–93–CSE–2, Electrical and Computer Engineering Department, University of Massachusetts, Amherst, Jan. 1993.

[14] N. Weste, and K. Eshraghian, *Principles of CMOS VLSI Design, A Systems Perspective*. Addison Wesley, 1988.

[15] M. Mehta, V. Parmar and E. Swartzlander, “High–Speed multiplier design using multi–input counter and compressor circuits,” in *Proc. of the 10th Symposium on Computer Arithmetic*, pp. 43–50, 1991.

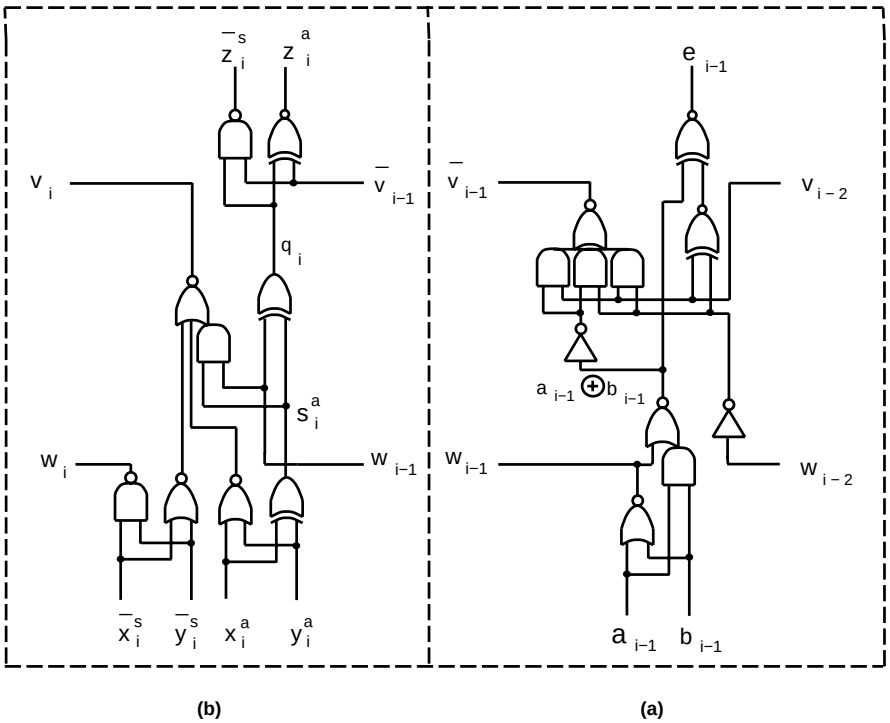

Figure 1 : Redundant binary adder cells at (a) unsigned digit position (b) signed digit position.

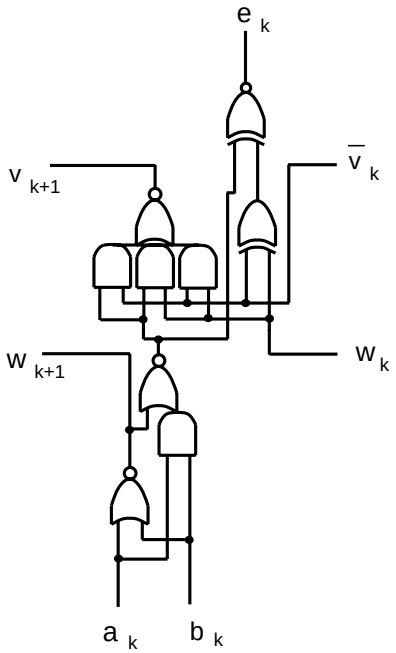

Figure 2 : Alternate cell for the unsigned digit position.

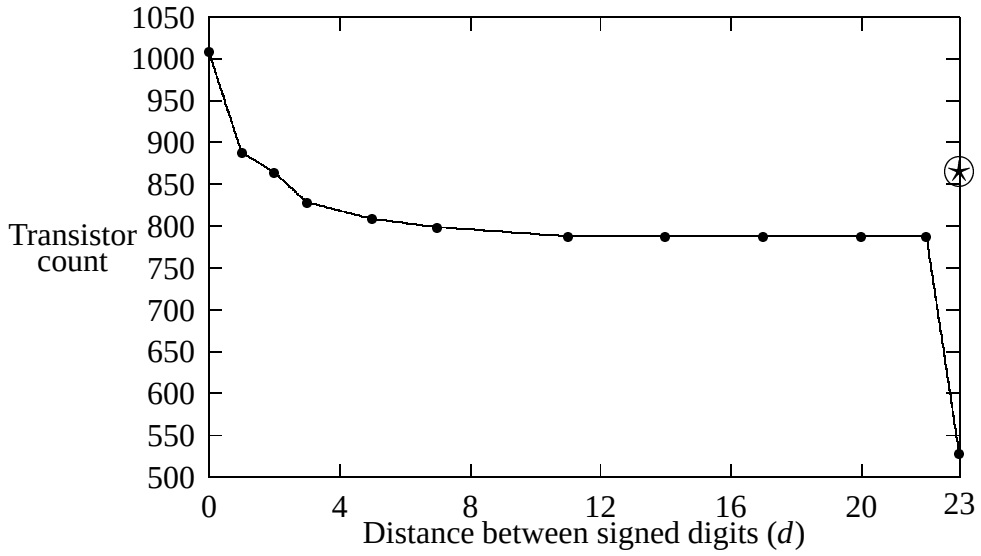

Figure 3(a) : Area (transistor count) vs. the distance d between two consecutive signed digits. $d = 0$ corresponds to the SD adder and $d = 23$ corresponds to an ordinary ripple–carry adder. The point denoted by the symbol " ⊛" at $d = 23$ corresponds to the carry-look-ahead adder with a blocking factor (i.e., look-ahead tree fan-in) of 4.

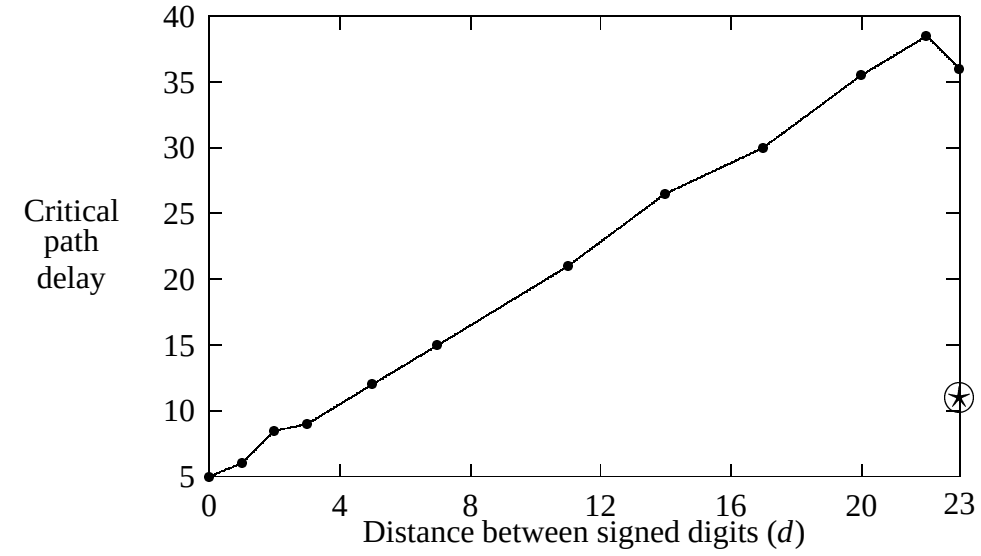

Figure 3(b) : Critical path delay vs. the distance d between two consecutive signed digits.

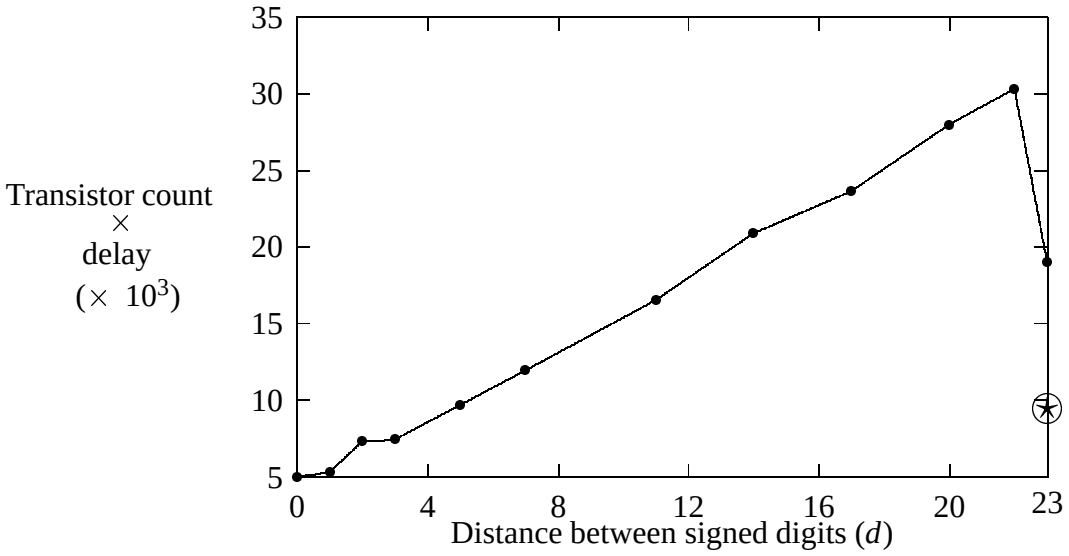

Figure 3(c) : Area × Time vs. the distance d between consecutive signed digits.

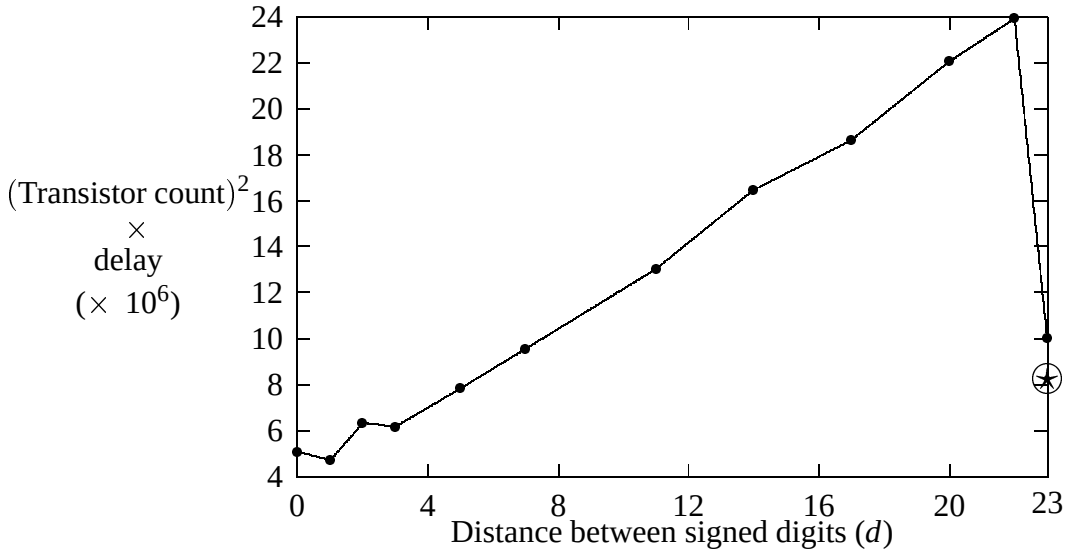

Figure 3(d) : (Area^2× Time) vs. the distance d between consecutive signed digits.

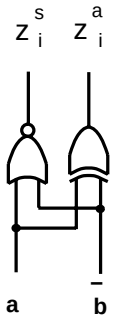

Figure 4 : Cell used at the top level of an SD tree to add operands in two's complement format and produce signed digit output [5].

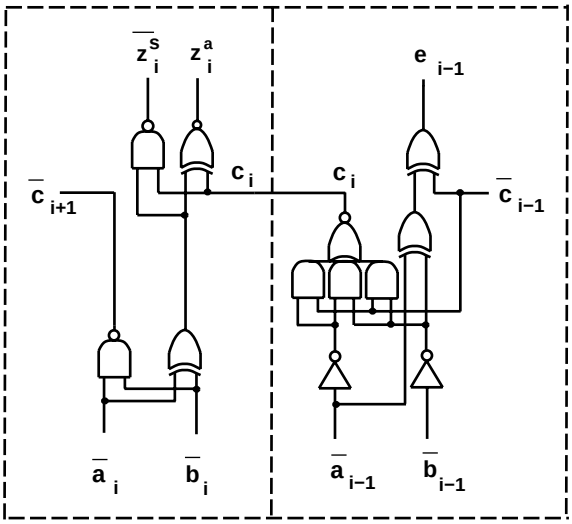

Figure 5 : Cells used (at the top level of an HSD tree) to add operands in two's complement format and produce HSD format output. (a) cell in a signed digit position (b) Cell in an unsigned digit position.