Intermediate Variable Encodings that Enable Multiplexor-Based Implementations of Two Operand Addition

Dhananjay S. Phatak
Electrical Engineering Department
State University of New York, Binghamton, NY 13902–6000
phatak@ee.binghamton.edu

I. Koren
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
koren@euler.ecs.umass.edu

(Proceedings of the IEEE ARITH'14, Adelaide, Australia, April 1999, pp 22–29)

Abstract

In two operand addition, bit-wise intermediate variables such as the "propagate" and "generate" terms are defined/evaluated first. Basic carry propagation recursion is then expressed in terms of these variables and is "unrolled" to obtain a tree structure for fast execution. In CMOS VLSI technology, multiplexors are fast and efficient to implement. Hence, we investigate in this paper all possible two-bit encodings for the intermediate variables and identify the ones that enable multiplexor-based implementations. Some of these encodings enable further simplification of the multiplexor-based realizations. Our analysis also shows that adopting an intermediate signed-digit representation simply amounts to selecting one of the possible encodings. Thus, there is no inherent advantage to the use of intermediate signed-digit representations in a two operand addition. Finally, we extend our analysis to the generalized look-ahead-recursions proposed by Doran.

1. Introduction

In a two operand addition of the numbers $A = \{a_{n-1}, \cdots a_i, \cdots a_0\}$ and $B = \{b_{n-1}, \cdots b_i, \cdots b_0\}$, bit-wise terms such as P_i and G_i are generated first and the basic carry propagation recursion is expressed in terms of these intermediate variables:

$$\begin{aligned} c_i &= G_i + P_i c_{i-1} \quad \text{where} \quad G_i = a_i \cdot b_i \\ &\quad \text{and} \;\; P_i \in \{(a_i + b_i), (a_i \oplus b_i)\} \end{aligned} \tag{1}$$

where "+" indicates logical OR, "$\cdot$" (or a product term) indicates logical AND, $\oplus$ denotes XOR and c_{i-1}, c_i denote the carries into and out-of position i, respectively. To enable fast execution, this first order (Boolean) recursion is unrolled, leading to a tree structure with logarithmic delay. This is achieved via defining group $P_{i:j}$ and $G_{i:j}$ variables for a group of bits $[i:j]$ $i \geq j$ in terms of the bit-wise signals P_i and G_i and then expressing recursions to synthesize P and G signals of bigger groups from those of smaller adjacent or overlapping subgroups [4]. These recursions can be summarized via the fundamental carry operation [1, 4, 7], fco , denoted by "$\odot$", and defined by

$$(P,G) \odot (\tilde{P},\tilde{G}) = (P \cdot \tilde{P}, G + P \cdot \tilde{G}) \tag{2}$$

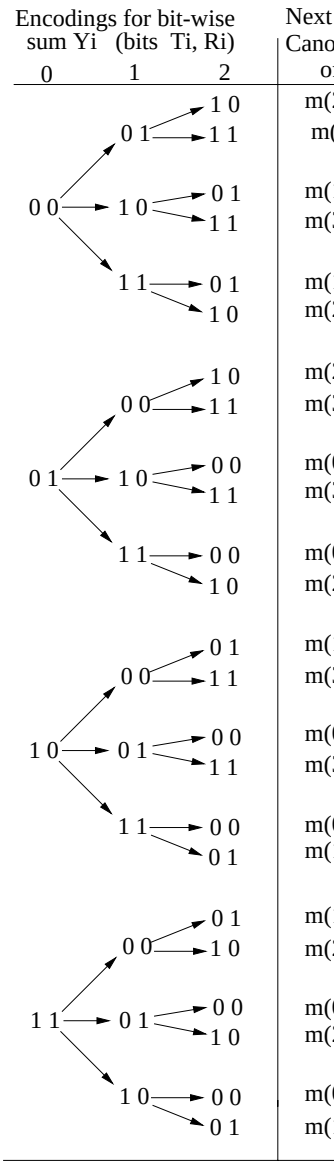
$$(P_{i:j}, G_{i:j}) = (P_{i:m}, G_{i:m}) \odot (P_{v:j}, G_{v:j}) \tag{3}$$

$$\text{where} \quad i \geq m;\; v \geq m-1 \;\text{ and }\; i \geq v > j$$

$$\text{and} \;\; c_i = G_{i:j} + P_{i:j} \cdot c_{j-1} \tag{4}$$

In the following section, we examine all possible encodings (with two bits or Boolean variables) for the intermediate variables and identify the ones that are suitable for multiplexor-based implementations. This analysis shows that adopting an intermediate signed-digit representation (in a two operand addition) simply amounts to selecting one of the possible encodings. This is demonstrated via a one-to-one correspondence between all possible encodings for bit-wise (result indicator) variables in conventional adders and those that go through an intermediate signed digit representation. We then extend our discussion to include the generalized look-ahead recursions which were proposed by Doran [2].

Given the above goals, the paper mainly deals with the underlying theory. Full hardware realizations are not considered and as a result, raw nanosecond and area values are less relevant in this exposition.



Next carry/borrow Ci as a Canonical SOP expression of (Ci-1, Ti, Ri)	Output sum-bit di	Correspondence
m(2, 5, 6) + d(3, 7)	m(1, 4, 6) + d(3, 7)	E0 <----> SD14
m(3, 5, 7) + d(2, 6)	m(1, 4, 7) + d(2, 6)	E1 <----> SD20
m(1, 5, 6) + d(3, 7)	m(2, 4, 5) + d(3, 7)	E2 <----> SD8
m(3, 6, 7) + d(1, 5)	m(2, 4, 7) + d(1, 5)	E3 <----> SD22
m(1, 5, 7) + d(2, 6)	m(3, 4, 5) + d(2, 6)	E4 <----> SD10
m(2, 6, 7) + d(1, 5)	m(3, 4, 6) + d(1, 5)	E5 <----> SD16
m(2, 4, 6) + d(3, 7)	m(0, 5, 6) + d(3, 7)	E6 <----> SD12
m(3, 4, 7) + d(2, 6)	m(0, 5, 7) + d(2, 6)	E7 <----> SD18
m(0, 4, 6) + d(3, 7)	m(2, 4, 5) + d(3, 7)	E8 <----> SD2
m(3, 6, 7) + d(0, 4)	m(2, 5, 7) + d(0, 4)	E9 <----> SD23
m(0, 4, 7) + d(2, 6)	m(3, 4, 5) + d(2, 6)	E10 <----> SD4
m(2, 6, 7) + d(0, 4)	m(3, 5, 6) + d(0, 4)	E11 <----> SD17
m(1, 4, 5) + d(3, 7)	m(0, 5, 6) + d(3, 7)	E12 <----> SD6
m(3, 4, 7) + d(1, 5)	m(0, 6, 7) + d(1, 5)	E13 <----> SD19
m(0, 4, 5) + d(3, 7)	m(1, 4, 6) + d(3, 7)	E14 <----> SD0
m(3, 5, 7) + d(0, 4)	m(1, 6, 7) + d(0, 4)	E15 <----> SD21
m(0, 4, 7) + d(1, 5)	m(3, 4, 6) + d(1, 5)	E16 <----> SD5
m(1, 5, 7) + d(0, 4)	m(3, 5, 6) + d(0, 4)	E17 <----> SD11
m(1, 4, 5) + d(2, 6)	m(0, 5, 7) + d(2, 6)	E18 <----> SD7
m(2, 4, 6) + d(1, 5)	m(0, 6, 7) + d(1, 5)	E19 <----> SD13
m(0, 4, 5) + d(2, 6)	m(1, 4, 7) + d(2, 6)	E20 <----> SD1
m(2, 5, 6) + d(0, 4)	m(1, 6, 7) + d(0, 4)	E21 <----> SD15
m(0, 4, 6) + d(1, 5)	m(2, 4, 7) + d(1, 5)	E22 <----> G3
m(1, 5, 6) + d(0, 4)	m(2, 5, 7) + d(0, 4)	E23 <----> SD9

Table 1: All possible encodings of the sum (or difference) y_i with bits (T_i, R_i).

2. Encodings

The conventional bit-wise propagate and generate variables can be viewed as an encoding of the bit-wise (algebraic) sum $y_i = a_i + b_i$ which can assume one of the three values $\{0,1,2\}$.

Two bits are sufficient to encode these three values. It is possible to use more than two bits to encode all the *bit-wise information of interest* that is necessary to be propagated. However, in this paper we only consider schemes that use two bits to encode the bit-wise information. Let the variables used in the ith bit position be labeled T_i and R_i to distinguish them from P_i and G_i. In fact, when P_i is defined to be the bit-wise OR of a_i and b_i, and G_i is the AND (as is conventionally done), it is just one instance of all possible encodings wherein (P_i, G_i) = "00" represents a sum (y_i) value of 0, (P_i, G_i) = "10" represents $y_i = 1$ and "11" encodes $y_i = 2$.

In all, there are 24 possible encodings that can be used to represent the sum y_i. All 24 encodings are summarized in Table 1. We examine all of them to determine the ones that enable multiplexor-based implementations. Each row of Table 1 shows one encoding which is labeled in the first

sub-column of the last column (titled "Correspondence") by labels E_r, $r = 0, 1, \cdots, 23$. For convenience, encodings are labeled from 0 onwards. The first (leftmost) multi-column (consisting of three sub-columns) shows the (T_i, R_i) encoding used to represent the three possible values of the sum y_i (0, 1, and 2). The next column shows the *canonical* sum-of-products (SOP) Boolean expression for the carry-out (c_i) in terms of the incoming carry c_{i-1} and the intermediate bit-wise signals T_i and R_i (determined by the encoding in that row). In the SOP expressions, the decimal labeling of minterms corresponds to variable order (c_{i-1}, T_i, R_i), and the notation $m(i, j, k) + d(n, l)$ indicates that the SOP expression contains minterms i, j, k and its value is a don't-care corresponding to minterms n and l.

The next column shows the logical SOP expression for the final sum output bit d_i in terms of (c_{i-1}, T_i, R_i) for the encoding in that row.

The don't cares in columns 2 and 3 (the d(k, l) terms in the SOP expressions, where $0 \leq k, l \leq 7$) arise because only three out of the four combinations of two bits (T_i, R_i) are needed to encode all possible values of y_i.

As an example, the 9th row shows encoding E8 in which $y_i = 0$ is encoded by (T_i, R_i) = "01", $y_i = 1$ is encoded by "10" and $y_i = 2$ is encoded by "00". Under this encoding, the canonical SOP expression for the carry propagation equation is $c_i = m(0, 4, 6) + d(3, 7)$.

Encoding E2 is the conventional carry generate and propagate encoding with $P = a_i \oplus b_i$, while E3 is the conventional carry generate and propagate encoding with $P = a_i + b_i$.

Table 1 is also used to demonstrate the equivalence between conventional adders and those based on an intermediate signed–digit representation. The one-to-one correspondence is explained in the next section.

To obtain a compact multiplexor-based look-ahead circuit, the carry-out c_i (which is a function of c_{i-1}, T_i and R_i) must be expressible in the form

$$c_i = \overline{S_i}\alpha_i + S_i c_{i-1} \tag{5}$$

This expression has the following desirable properties:
(i) The incoming carry c_{i-1} is the late arriving signal and therefore any expression for c_i should involve only c_{i-1}, not its complement (to avoid extra delay in the inversion). Likewise, c_i should be available in uncomplemented form (note that fully restoring complementary static CMOS is inherently "inverting" logic). Finally, it should be possible to input c_{i-1} via the shortest (delay) path. All of this is possible in a transmission-gate-based multiplexor realization of equation (5) above.
(ii) For the same reason, the load on signal c_{i-1} should be as small as possible. In a transmission gate multiplexor implementation of the above equation, c_{i-1} sees only the drain or source capacitance which is likely to be smaller than a gate capacitance.
(iii) The multiplexor control signals S_i and the signal α_i should be derived from the (fewest possible) operand bits, so that they can be realized with small delay and hardware.
Henceforth in this paper, expressions like the right hand side of equation (5) are said to be in the "mux-form".

Having identified the above desirable attributes, the next logical question is: which of the 24 encodings in Table 1 lead to expressions of the form (5) (which in turn lead to recursions of the form in equation (11); making it possible to synthesize a look-ahead-tree using multiplexors as explained later)?

It can be verified that out of the 24 encodings in Table 1, the following 16 encodings, viz.,

L = {E0, E2, E4, E5, E7, E8, E9, E10, E13, E14, E15, E16, E18, E19, E21, E23}

allow a multiplexor-based implementation of the form (5) where the variable S_i is either an XOR or XNOR of the operand bits. We illustrate this for encoding E0 for which the multiplexor-based expression for c_i is

$$c_i = \overline{R}_i T_i + R_i c_{i-1} \tag{6}$$

For this encoding, the T_i, R_i are generated from the truth table in Table 2.

a_i	b_i	$y_i = a_i + b_i$	T_i	R_i	Further simplification of T_i
0	0	0	0	0	0
0	1	1	0	1	$\times$
1	0	1	0	1	$\times$
1	1	2	1	0	1

Table 2: Derivation of T_i and R_i for Encoding E0 in Table 1.

From Table 2 we obtain $R_i = a_i \oplus b_i$ and $T_i = a_i \cdot b_i$.
In the last column we show further simplification of the T_i variable: note that R_i is well defined for all combinations of input operand bits (no don't cares). This combined with equation (6) indicates that whenever $R_i = 1$, the incoming carry is propagated and the value of T_i is inconsequential (since it is ANDed with $\overline{R}_i$ which is 0). Hence, whenever $R_i = 1$, the actual value of T_i can be replaced by a don't care (denoted by $\times$) as shown in the last column of Table 2. With these don't cares, T_i can be further simplified to

$$T_i = a_i \quad \text{or} \quad T_i = b_i \tag{7}$$

Thus the conventional bit-wise AND (G_i) can be replaced by one of the operand bits itself, without affecting the rest of the carry look-ahead tree and the adder structure.

Out of 16 encodings that enable multiplexor-based implementations of the look-ahead recursion, eight encodings, viz.,

L_1 = {E0, E2, E4, E5, E7, E9, E13, E15}

make it possible to simplify T_i as indicated above and replace it by one of the operand bits. The set of these eight encodings is henceforth denoted by L_1.

In the remaining eight encodings, viz.

L_2 = {E8, E10, E14, E16, E18, E19, E21, E23}

T_i can be simplified and set equal to either $\overline{a}_i$, or $\overline{b}_i$, i.e., the complement of one of the operand bits.

The next question of interest is: can the mux-form recursions in equations (5) and (6) be extended to groups of bits so that a look-ahead-tree of multiplexors can be constructed; and which of the 24 encodings lead to such trees?

It turns out that all the 16 encodings in the set L lead to multiplexor-based look-ahead trees. For a succinct presentation, it is useful to introduce the *"fundamental selection operator"* (fso) denoted by "$\otimes$" and defined by

$$(R,T) \otimes (\tilde{R},\tilde{T}) = (R \cdot \tilde{R},\ \ T \cdot \overline{R} + \tilde{T} \cdot R) \tag{8}$$

which is analogous to the well-known fundamental carry operator (explained in the previous section). Note that the above relation (8) includes *two* equations, and one of them (the second) has the mux-form. It turns out that all mux-form recursions of interest can be expressed in terms of the fso operator, $\otimes$, as shown next.

Starting with bit-wise variables, let

$$P_i^{\oplus} \quad = \quad a_i \oplus b_i \quad \text{and}$$

$$Q_i \quad \in \quad \{a_i, b_i, G_i\} \quad \text{where} \quad G_i = a_i \cdot b_i \tag{9}$$

i.e., Q_i can take any of the 3 values a_i, b_i, G_i.

Then, it can be shown that all eight encodings in the set L_1 lead to the following recursions:

$$c_i \quad = \quad \overline{P_i^{\oplus}} Q_i + P_i^{\oplus} c_{i-1} \tag{10}$$

$$(P_{i:j}^{\oplus}, Q_{i:j}) \quad = \quad (P_{i:m}^{\oplus}, Q_{i:m}) \otimes (P_{v:j}^{\oplus}, Q_{v:j}) \tag{11}$$

$$\text{where} \qquad i \geq m;\ v \geq m-1 \ \text{ and } \ i \geq v > j$$

$$c_i \quad = \quad \overline{P^{\oplus}}_{i:j} \cdot Q_{i:j} + P_{i:j}^{\oplus} \cdot c_{j-1} \tag{12}$$

The $P_{i:j}^{\oplus}$ and $Q_{i:j}$ in the above expressions, are similar to the group propagate and generate variables in the conventional fco operation. In particular, the equations summarized by relation (11) show that the P and Q signals for a group of bits $[i:j]$ can be synthesized from the P and Q signals of two adjacent or *overlapping* subgroups of bits $[i:m]$ and $[v:j]$ using multiplexor-based selection operation (fso) which can be employed throughout a look-ahead-tree.

If the bit-wise P variable is restricted to be the XOR of the operand bits and the bit-wise Q variable is restricted to be the AND, it can be shown that the operators fco ($\odot$) and fso ($\otimes$) are completely interchangeable, so that they can be arbitrarily mixed-and-matched as required. In such a case, one could realize the first few levels of the look-ahead tree with the fso operation and then switch over to the conventional fco operation if required (to avoid cascading too many transmission gates in series, for example).

Even if one starts off with $Q_i = a_i$ (or $Q_i = b_i$) and the fso operation, it is still possible to switch over to the conventional fco operation at any intermediate level of the look-ahead tree with a small hardware and delay overhead (associated with one complex gate operation).

While specific instances of multiplexor-based look-ahead-trees have been investigated in the literature (for instance [10], [12], etc.) the contributions of this paper are:

(i) A unified treatment and exploration of all possible two-bit encodings to identify the ones that enable multiplexor-based look-ahead-tree implementation.

(ii) Simplification of the variable (Q_i) and inter-operability of the fco and fso functions with and without these Q_i simplifications; and

(iii) Demonstration that adopting an intermediate signed-digit representation simply amounts to selecting one of the 24 possible encodings, which is done in the next section.

Note that Manchester and carry-skip methods in effect employ recursions (10)–(12) above. In fact, most implementations, classic [1] as well as recent ones [3, 7, 8, 11, 12, 13], in effect employ the recursions in (1)–(4) or (10)–(12) where Q_i is restricted to be the conventional $G_i = a_i \cdot b_i$.

3. Correspondence between Adders Based on Intermediate Signed-Digit Representations and Conventional Ones

Several algorithms for addition that employ an intermediate Signed-Digit (SD) representation have been proposed (for instance [5, 12]). In the following we show that these seemingly different methods simply amount to using a different encoding (from among the 24 shown in Table 1). In these methods, $(A+B)$ is re-written as

$$A + B = (A - \overline{B} - 1) \text{ modulo } 2 \tag{13}$$

where $\overline{B}$ is the one's complement of B, obtained by inverting all the bits of B:

$$\overline{B} = 2^n - 1 - B \quad \text{, where } n \text{ is the word-length} \tag{14}$$

The -1 in equation (13) can be taken care of by forcing a carry (borrow)-in $c_{\text{in}} = -1$. The modulo operation simply amounts to discarding the outgoing borrow, except when there is an overflow, which can be detected as in conventional addition. Assuming

(i) c_{n-1} and c_n represent the borrow-in and borrow-out of the most significant bit position, and

(ii) a logical "1" is used to represent a borrow of algebraic value -1 (which implies that a logical "0" indicates no borrow or a borrow of value 0);

it can be shown that *overflow* $= c_n \oplus c_{n-1}$ (15)

as in conventional addition.

Since each of the bits of A and $\overline{B}$ can be 0 or 1, a bit–wise subtraction directly leads to a signed-digit output

representing $(A-\overline{B})$, where each digit $y_i = a_i - \overline{b}_i$ is in the range $\{-1,0,1\}$. The bit–wise subtraction can be carried out in parallel for all the digit positions because there is no need to propagate signals from one digit position to the next.

The bit-wise difference y_i requires two variables for its encoding. Let the variable pair used to encode the three values of y_i be (T_i,R_i). After these intermediate variables are evaluated, a "borrow" c_i is propagated from the least significant bit to the most significant bit. The algebraic value of c_{in} is -1. Likewise, borrow c_i into position i has a negative weight, i.e., $c_i \in \{-1,0\}$ for $i \geq 0$.
The final operation at each of the digit positions is

$$w_i = y_i - c_{i-1}. \tag{16}$$

which can result in any of the four values $\{-2,-1,0,1\}$. Here, w_i is recoded with two bits:
(1) c_i representing the "outgoing borrow",
(2) the final sum output bit d_i
that satisfy the relation

$$w_i = y_i - c_{i-1} = d_i - 2c_i \tag{17}$$

Note that in a conventional addition, the analogous operation at each digit position is $w_i = y_i + c_{i-1}$ and hence w_i can assume any of the four values {0,1,2,3}, which are recoded with two bits: (1) c_i representing the "outgoing carry", and (2) the final sum output bit d_i. These two bits satisfy the relation

$$w_i = y_i + c_{i-1} = d_i + 2c_i \tag{18}$$

The encoding of the final sum-output bit d_i is fixed because the final sum output must be in **non redundant** two's complement format. As a result, equations (17) and (18) uniquely determine *the algebraic values of* c_i and d_i for each possible value of w_i (abbreviated $w_i \leftrightarrow (c_i,d_i)$) as indicated below:

$-2 \leftrightarrow (-1,0)$; $-1 \leftrightarrow (-1,1)$; $0 \leftrightarrow (0,0)$ and $1 \leftrightarrow (0,1)$

for adders based on intermediate SD representation; whereas

$0 \leftrightarrow (0,0)$; $1 \leftrightarrow (0,1)$ $2 \leftrightarrow (1,0)$; $3 \leftrightarrow (1,1)$

for conventional adders.

The only difference that can arise between the two schemes (conventional versus those that employ intermediate SD representation) is therefore due to the encoding used to represent the intermediate result y_i and the resultant Boolean recursion equations.

There are 24 possible encodings to represent $y_i \in \{-1,0,1\}$ with two bits. For every encoding (and consequently for every recursion) in a scheme that utilizes intermediate SD result, there is a corresponding encoding (and a corresponding recursion) in the conventional scheme. This correspondence is summarized in Table 1.

The first column in Table 1 indicates the 24 possible encodings that can be used to represent the three values which the intermediate bit-wise result y_i can assume: $\{-1,0,1\}$ in adders based on intermediate signed digits (please refer to the footing at the bottom of the first column in Table 1); and $\{0,1,2\}$ in conventional adders. The three sub-columns in the first column list the bit values of the ordered pair (T_i,R_i) for y_i equal to -1, 0 and +1, respectively, in a scheme based on intermediate signed digits. The same three sub-columns also represent the bit values of the pair (T_i,R_i) for y_i equal to 0, 1 and 2, respectively, in a conventional addition. For instance, the first row corresponds to an encoding where the pair (T_i,R_i) = (0,0) represents the intermediate result $y_i = -1$, the pair (0,1) represents $y_i = 0$ and the pair (1,0) represents $y_i = +1$. The same row also shows an encoding for conventional addition where the bit pair 00 represents the algebraic value 0, the pair 01 represents the value 1 and 10 represents the value 2. Encodings for adders based on signed digits have labels SDr where r = $0,1,\cdots,23$. Thus the first row of the table specifies encoding E0 for conventional adders and encoding SD0 for adders based on intermediate SD result. The last column indicates the one-to-one correspondence between the two schemes which is explained next.

The first correspondence (in the first row) indicates that E0 and SD14 are equivalent. Encoding SD14 corresponds to the 15th row of the table, showing that under this encoding (for adders based on interim SD representation), the intermediate bit-wise difference value $y_i = -1$ is encoded by (T_i,R_i) = (1,0); $y_i = 0$ is encoded by "01" while $y_i = +1$ is encoded by "00". Here, c_i represents the "borrow-out" of position i while d_i denotes the final output bit at position i. With this encoding, it can be verified that

$$c_i = f(c_{i-1},T_i,R_i) = m(2,5,6) + d(3,7) \tag{19}$$

$$d_i = f(c_{i-1},T_i,R_i) = m(0,2,5) + d(3,7) \tag{20}$$

Note that the canonical SOP expression (19) for the borrow-out (under this encoding, viz. SD14) is identical to the expression for the carry-out under encoding E0 for conventional adders (as seen in the first row of Table 1). The expression for d_i under encoding SD14 is the complement of the expression for d_i under E0 (this happens because one of the operand bits is complemented before generating the bit-wise difference, in the scheme based on intermediate SD representation).

All the equivalences ($\text{E}_k \longleftrightarrow \text{SD}_m$) listed in the last column of Table 1 hold in the same sense, viz.,

(i) Canonical SOP expressions for c_i are identical under encoding SD_m for adders based on intermediate SD representation, and encoding E_k for conventional adders; while

(ii) The expression for d_i under encoding SD_m is the complement of the corresponding one under encoding E_k.

The encodings turn out to be symmetric in the sense that $\text{E}_i \longleftrightarrow \text{SD}_j$ and $\text{E}_j \longleftrightarrow \text{SD}_i$ for $0 \leq i,j \leq 23$.

In view of this one-to-one correspondence, it is readily seen that 16 out of the 24 SD encodings viz.

M = {SD0, SD2, SD4, SD5, SD7, SD8, SD9, SD10, SD13, SD14, SD15, SD16, SD18, SD19, SD21, SD23}

also allow multiplexor-based implementations like the conventional adders.

Similarly, exactly half of these sixteen encodings enable further simplification of T_i to equal one of the operand bits a_i or b_i. These are the encodings

M_1 = {SD8, SD10, SD14, SD16, SD18, SD19, SD21, SD23}.

In the remaining eight encodings viz.,

M_2 = {SD0, SD2, SD4, SD5, SD7, SD9, SD13, SD15}

T_i can be simplified to equal the complement of one of the two operand bits (requiring an inverter).

It is likely that a similar one-to-one correspondence also exists between encodings of intermediate variables in conventional addition and stored-carry/borrow or carry-sum [9] representations.

4. Other Variants of Carry look-ahead Addition

The outgoing carry, c_i, is a meaningful signal/variable and hence is the variable of choice for expressing the fundamental carry recursion. However, in theory, other Boolean variables can be defined and propagated in the look-ahead tree, instead of the carry variable. Doran [2] investigated all possible Boolean variables that can be used to express the fundamental carry recursion and showed a methodology to systematically derive the recursive Boolean equations when expressed in terms of the chosen variables. The main goal of his work was to identify forms of recursion which enable more compact and faster implementation of the look-ahead signal generation for a group of four bits. This was inspired by Ling's adder [6] in which instead of propagating the carries, Ling defines

$$H_i = c_i + c_{i-1} \tag{21}$$

i.e., $H_i = 1$ if there is a carry in or out of position i. One possible interpretation of H_i is that it is true if "something interesting" happens at position i. Ling's adder propagates this variable H by expressing fundamental carry recursions in terms of H and operand bits in positions i and $(i-1)$ and evaluates the sum output bit d_i in terms of this variable H [2, 6].

$$H_i = G_i + Z_{i-1}H_{i-1} \quad \text{where,} \tag{22}$$
$$G_i = a_i \cdot b_i \quad \text{and} \quad Z_i = a_i + b_i$$
$$d_i = Z_i \oplus H_i + G_i Z_{i-1} H_{i-1} \tag{23}$$

In contrast, the conventional adder implements

$$c_i = G_i + P_i c_{i-1} \quad \text{where} \tag{24}$$
$$G_i = a_i \cdot b_i \quad \text{and} \quad P_i = a_i \oplus b_i$$
$$d_i = P_i \oplus c_{i-1} \tag{25}$$

The only difference between the recursions (in the c and H variables) lies in the coefficient of the propagated variable. In the conventional case, the coefficient of c_{i-1} (in equation (24)) is derived solely from operands from bit position i, whereas in Ling's case the coefficient of H_{i-1} (in equation (22)) is derived only from operands in position $(i-1)$. This seemingly small difference leads to a substantial advantage when the recursion is unrolled to four bit positions to evaluate the group H and G variables:

$$G_{i:i-3} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + P_i P_{i-1} P_{i-2} G_{i-3} \tag{26}$$

$$H_{i:i-3} = G_i + Z_{i-1}G_{i-1} + Z_{i-1}Z_{i-2}G_{i-2} + Z_{i-1}Z_{i-2}Z_{i-3}G_{i-3}$$
$$= G_i + G_{i-1} + Z_{i-1}G_{i-2} + Z_{i-1}Z_{i-2}G_{i-3} \quad \text{since } Z \cdot G = G \tag{27}$$

It is seen that $H_{i:i-3}$ requires a smaller expression, making it faster to evaluate than $G_{i:i-3}$. The expression for the final sum output bit d_i in Ling's scheme is more complex, but that penalty was deemed worthwhile (for the implementation technology which was available at that time) in exchange for speeding up the look-ahead signal generation for a group of four bits.

Doran identified the desirable properties in Ling's recursion:

(i) Only H_{i-1} appears, not its complement (this avoids inversion delays).

(ii) The coefficient of H_{i-1} is derived from operands in bit position $(i-1)$ alone.

He then showed that H is not the only variable which has all these desirable properties. He systematically arrived at all possible recursions which have the same properties as Ling's recursion (and hence are equivalent from a hardware implementation viewpoint). These are reproduced for convenience in Table 3 (on the next page) where Adder 1 is Ling's adder.

The main difference from the conventional adders is that the bit-wise P, G or Z signals from two positions, i and $(i-1)$, are used in the fundamental recursion (in row 1 of Table 3). The issue of interest is whether any of these recursions lend themselves to mux-form expressions (such as those in equations (5) or (11)) which can be implemented by a single multiplexor.

To this end, a cascaded mux-form representation (one of the several possible) is shown below for each of the four recursions in Table 3.

$$\text{Adder1: } X_i = G_i \cdot 1 + \overline{G}_i(\overline{P}_{i-1}Q_{i-1} + P_{i-1}X_{i-1}) \tag{28}$$
$$\text{Adder2: } X_i = \overline{P}_i \cdot 1 + P_i(\overline{P}_{i-1}Q_{i-1} + P_{i-1}X_{i-1}) \tag{29}$$
$$\text{Adder3: } X_i = \overline{Z}_i \cdot 1 + Z_i(G_{i-1} \cdot 0 + \overline{G}_{i-1}X_{i-1}) \tag{30}$$
$$\text{Adder4: } X_i = \overline{P}_i \cdot 1 + P_i(G_{i-1} \cdot 0 + \overline{G}_{i-1}X_{i-1}) \tag{31}$$

	Adder 1	Adder 2	Adder 3	Adder 4
Recursion: $X_i =$	$G_i + Z_{i-1}X_{i-1}$	$\overline{P}_i + Z_{i-1}X_{i-1}$	$\overline{Z}_i + \overline{G}_{i-1}X_{i-1}$	$\overline{P}_i + \overline{G}_{i-1}X_{i-1}$
Relation to carries: $X_i =$	$G_i + c_{i-1}$	$\overline{P}_i + c_{i-1}$	$\overline{Z}_i + \overline{c}_{i-1}$	$\overline{P}_i + \overline{c}_{i-1}$
Sum output bit: $d_i =$	$Z_i \oplus X_i$ + $G_i Z_{i-1} X_{i-1}$	$\overline{X}_i$ + $\overline{P}_i Z_{i-1} X_{i-1}$	$(G_i \oplus X_i)$ · $\overline{\overline{Z}_i \overline{G}_{i-1} X_{i-1}}$	X_i · $\overline{\overline{P}_i \overline{G}_{i-1} X_{i-1}}$

Table 3: The four recursions derived by Doran [2].

where, Q_{i-1} can take any of the three values $\{a_{i-1}, b_{i-1}, G_{i-1}\}$.

Equation (28) was obtained by starting from the definition $X_i = G_i + c_{i-1}$. From this, it is possible to obtain the expression $X_i = G_i + G_{i-1} + Z_{i-1}c_{i-1}$. Substituting $c_{i-1} = Z_{i-1}X_{i-1}$, along with $G_{i-1} = \overline{P}_{i-1}G_{i-1}$ and $Z_{i-1} = P_{i-1}Z_{i-1}$ yields the result. Equation (29) can be derived likewise.

Each of the above equations (28)–(31) can be implemented as a cascade of two muxes, where the expression in parenthesis is implemented by the first multiplexor, whose output is one of the select lines of the second multiplexor. In CMOS, it is possible (and might be advantageous) to use AOI, OAI and complex gates instead of multiplexors; and directly implement the fundamental recursion expressions in Table 3. However, the main issue we are addressing is not finding out the most efficient implementation of Doran's recursions. Rather, the relevant question is whether Doran-style recursions can be realized using a *single* MUX (if not, then these recursions don't "fit" as well with MUX trees as some other recursions identified above). To the best of our knowledge, the answer is no.

This can be seen by exploring the generator equations underlying the above recursions. Following Doran's method of analysis, let

$$X_i = \psi(a_i, b_i)c_{i-1} + \phi(a_i, b_i)\overline{c}_{i-1} \tag{32}$$

be the generic expression for the recursion variable X. Since $A + B = B + A$, the functions $\psi(a_i, b_i)$ and $\phi(a_i, b_i)$ must be symmetric in a_i and b_i. There are eight symmetric Boolean functions of a_i and b_i which fall under two classes [2] as shown next:

$$\begin{array}{ll} f_1 = a_i\overline{b}_i + \overline{a}_i b_i = P_i + 0 & f_5 = \overline{P}_i = a_i b_i + \overline{a}_i\overline{b}_i \\ & \quad = \overline{f}_1 = \overline{P}_i \cdot 1 \\ f_2 = a_i + b_i = P_i + G_i & f_6 = \overline{a}_i\overline{b}_i = \overline{f}_2 = \overline{P}_i\overline{G}_i \\ f_3 = \overline{a}_i + \overline{b}_i = P_i + \overline{Z}_i & f_7 = a_i b_i = \overline{f}_3 = \overline{P}_i Z_i \\ f_4 = 1 = P_i + \overline{P}_i & f_8 = 0 = \overline{f}_4 = \overline{P}_i P_i \\ \text{Class I} \quad P_i \supset f_i & \text{Class II} \quad P_i \supset \overline{f}_i \end{array} \tag{33}$$

It can be said that $P_i \supset f_i, i = 1,2,3,4$, i.e., whenever $P_i = 1$, all the functions f_1, f_2, f_2, f_4 are also 1. Likewise, $P_i = 1$ implies that the remaining four functions f_5, f_6, f_7, f_8 all take the value 0 (f_4 and f_8 are constant functions and included only for the sake of completeness. obviously, they are not used anywhere).

Thus there are 64 pairs of ψ and ϕ functions and 64 possible recursions. Not all of these are useful: many of them do not form an adder. In other words, for these definitions of X_i, the final sum output bit cannot be retrieved from X_i and operand bits at position i ($\phi = \psi = 0$, yielding $X_i = 0$ is such an example). Furthermore, it can be shown that if ψ and ϕ belong to the same class (Class I or II in equation (33) above), then the resultant X_i does not form an adder [2]. That rules out 32 (16+16) possibilities. The remaining 32 recursions form adders (i.e., variable X_i can be utilized instead of the carry variable to express the fundamental carry recursions). These recursions fall into two classes whose generator equations are

$$\begin{array}{lrcll} \text{Class I:} & X_i & = & (P_i + u)c_{i-1} + v\overline{c}_{i-1} & (34) \\ & c_i & = & G_i + P_i X_i = G_i + Z_i X_i & (35) \\ & & = & G_i\overline{X}_i + Z_i X_i \qquad \text{and} & \\ \text{Class II:} & X_i & = & uc_{i-1} + (P_i + v)\overline{c}_{i-1} & (36) \\ & c_i & = & G_i + P_i\overline{X}_i = G_i + Z_i\overline{X}_i & (37) \\ & & = & G_i X_i + Z_i\overline{X}_i & \\ & \text{where,} & & u, v \in \{0, \overline{Z}_i, G_i, \overline{P}_i\} & (38) \end{array}$$

We illustrate the analysis method for the first recurrence only, the second recursion in equation (36) can be handled in an identical manner. The first generator equation (34) leads to the recursion

$$\begin{array}{rcl} X_i & = & (P_i + u)\left[Z_{i-1}X_{i-1} + G_{i-1}\overline{X}_{i-1})\right] \\ & & +v\left[\overline{Z}_{i-1}X_{i-1} + \overline{G}_{i-1}\overline{X}_{i-1}\right] \\ & = & \left[(P_i + u)Z_{i-1} + v\overline{Z}_{i-1}\right]X_{i-1} \\ & & +\left[(P_i + u)G_{i-1} + v\overline{G}_{i-1}\right]\overline{X}_{i-1} \end{array} \tag{39}$$

To enable multiplexor-based implementation, only X_{i-1} or its complement $\overline{X}_{i-1}$, but not both, should appear in the above recurrence. For illustration, consider the case in which X_{i-1} appears, and not it's complement. This happens only if the term in $\overline{X}_{i-1}$ can be absorbed into the term in X_{i-1} in equation (39) above, which in turn happens if

$\left[(P_i + u)G_{i-1} + v\overline{G}_{i-1}\right] \supset \left[(P_i + u)Z_{i-1} + v\overline{Z}_{i-1}\right]$

i.e., if

$$\left[(P_i+u)G_{i-1}+vP_{i-1}+v\overline{Z}_{i-1}\right] \supset \left[(P_i+u)G_{i-1}+(P_i+u)P_{i-1}+v\overline{Z}_{i-1}\right] \quad (40)$$

after substituting $\overline{G}_{i-1} = P_{i-1} + \overline{Z}_{i-1}$ and $Z_{i-1} = G_{i-1} + P_{i-1}$; the above holds if $vP_{i-1} \supset (P_i+u)P_{i-1}$ since G_{i-1}, P_{i-1} and $\overline{Z}_{i-1}$ are disjoint, i.e., only one of them can take the value 1. This condition finally leads to

$$v \supset P_i + u \quad \text{i.e.,} \quad v \supset u \quad \text{since} \quad u, v \supset P_i \quad (41)$$

This condition leads to the simplifications

$$X_i = v + (P_i+u)c_{i-1} \quad \text{and the recursion to}$$

$$\begin{aligned} X_i &= (P_i+u)G_{i-1} + v + [(P_i+u)Z_{i-1}+v]X_{i-1} \\ &= (P_i+u)G_{i-1} + v + [(P_i+u)Z_{i-1}]X_{i-1} \end{aligned} \quad (42)$$

The above expression can be implemented as a single multiplexor only if $(P_i+u)G_{i-1}$ can be expressed in the form

$$\overline{(P_i+u)Z_{i-1}} \cdot \mu \quad (43)$$

where μ can be some Boolean function of operand bits $\{a_i, b_i, a_{i-1}, b_{i-1}\}$. For the u and v choices dictated by constraints (38) and (41) it can be verified that an expression of the form required by (43) is not feasible, which leads to the conclusion that Doran's recursions cannot be expressed using a single multiplexor.

In other words, involving operands from two bit positions (i and $i-1$) in fundamental carry recursion expressions may be beneficial if complex gates are to be used in the hardware realization; however, it is not advantageous in multiplexor-based implementations.

5. Conclusion

We investigated all possible two-bit encodings for the intermediate bit-wise variables in two operand addition and identified the ones that enable fast and compact multiplexor-based implementations. This analysis shows that adopting an intermediate signed-digit representation simply amounts to selecting one of the possible encodings. This was demonstrated via a one-to-one correspondence between the two schemes. It is intuitively clear that carry-save, carry-sum, stored carry/borrow and other types of schemes will also turn out to be equivalent to conventional adders, i.e., every encoding in our table will correspond to some encoding in each of those schemes.

Doran has proposed a general framework for look-ahead recursions leading to several variants of look-ahead adders that were faster to implement (in the technology in-vogue at that time). We examined those variants and found that they do not enable efficient multiplexor-based implementations.

References

[1] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. on Computers*, TC-31(3):260–264, Mar. 1982.

[2] R. W. Doran. Variants of an improved carry look-ahead adder. *IEEE Trans. on Computers*, 37:1110–1113, Sept. 1988.

[3] V. Kantabutra. A recursive carry–look–ahead/carry–select hybrid adder. *IEEE Trans. on Computers*, 42(12):1495–1499, Dec. 1993.

[4] I. Koren. *Computer Arithmetic Algorithms*. Brookside Court Publishers, Amherst, Massachusetts, 1998.

[5] 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*, pages 80–86, 1987.

[6] H. Ling. High-speed binary adder. *IBM Journal of Research and Development*, 25:156–166, May 1981.

[7] T. Lynch and E. E. Swartzlander. A Spanning Tree Carry Lookahead Adder. *IEEE Trans. on Computers*, 41(8):931–939, Aug. 1992.

[8] M. Suzuki and Ohkubo, N., et. al. A 1.5-ns 32-b CMOS ALU in Double Pass-Transistor Logic. *IEEE Journal of Solid-State Circuits*, 28(11):1145–1150, Nov. 1993.

[9] C. Nagendra, R. M. Owens, and M. J. Irwin. Unifying Carry-Sum and Signed-Digit Number Representations. Technical Report CSE–96–036, Computer Science and Engineering Department, Pennsylvania State University, 1996.

[10] V. Oklobdzija. Simple and Efficient CMOS Circuit for Fast VLSI Adder Realization. In *Prof. of IEEE Int. Sym. on Circuits and Systems (ISCAS'88)*, volume I, pages 235–238, 1988.

[11] T. Sato, M. Sakate, H. Okada, T. Sukemura, and G. Goto. An 8.5-ns 112-b Transmission Gate Adder with a Conflict-Free Bypass Circuit. *IEEE Journal of Solid State Circuits*, 27(4):657–659, April 1992.

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

[13] 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*, SC-27(1):109–112, Jan. 1992.