MIRROR ARITHMETIC

Jean P. Chinal

Abstract

Mirror coding for signed numbers is defined by means of a set of primitive powers of two \( \{2^1, 2^{n-1}, \ldots, 2^0\} \) where signs of the usual set used in 2's complement representation are reversed. Use of the mirror representation is shown as an alternate design approach and illustrated by a special purpose adder design in mirror code, by an alternate proof of a basic property of signed-digit arithmetic and as another interpretation of cells used in some array multipliers for signed numbers. Lastly, the concept is used to define a variable mode redundant coding, allowing simple sign-flipping without overflow.

1. Principle

Given a signed number \( x \), its ordinary two's complement representation \( x' \) is a binary word \( x \) with at least four distinct interpretations. For a \((n+1)\)-digit \( x \) (with sign bit) let us write:

\[
x = a x_{n-1} \ldots x_1 \ldots x_0 \quad (a, x_i \in \{0, 1\})
\]

\[
x' = [0 x_{n-1} \ldots x_1 \ldots x_0]
\]

\[
x'' = a x_{n-1} \ldots x_1 \ldots x_0
\]

Also, let \( \nu(a) \) be the arithmetic value of Boolean \( a \) (arithmetic 1 for Boolean 1 and vice versa) and let us denote by \( x' \) and \( x'' \) the arithmetic values of \( x \), \( x' \) and of \( x'' \). If \( |x|_n \) is the residue of \( x \) modulo \( n \), we have:

\[
x = -\nu(a) 2^n + x'
\]

\[
x' = |x|_2^n
\]

\[
x'' = +\nu(a) 2^n + x'
\]

\[
x'' = |x|_2^n + x'_{n-1}
\]

\[
x = \nu(a)(-2^n) + \sum_{k=0}^{n-1} \nu(x_k) 2^i
\]

Thus \( x \) may be seen as a residue modulo \( 2^n \) with appended sign \( a \) on the left, as a residue modulo \( 2^{n+1} \) as a special signed-digit number (digit range \( \{-1,0,1\} \)) where the \(-1\) would be restricted to the leftmost position only or, still \((3,7)\) of a weighted sum with coefficients in \( \{0,1\} \) but with weights taken from an inventory of powers of two of both signs, namely

\[
I = \{-2^1, 2^{n-1}, \ldots, 2^1, \ldots, 2^0\}
\]

\[
I = \{+2^n, -2^{n-1}, \ldots, +2^1, \ldots, -2^0\}
\]

Here, all powers except the highest, are negative. Let us use this set ("mirror set") to represent signed numbers, i.e. let us look for digits \( \hat{x} \), such that

\[
\hat{x} = a \hat{x}_{n-1} \ldots \hat{x}_1 \ldots \hat{x}_0 \quad (a, \hat{x}_i \in \{0, 1\})
\]

\[
\nu(\hat{x}) = \nu(x') 2^{n-1} - \nu(x'_{n-1}) 2^{n-2} - \nu(x_1) 2^0
\]

where \( \hat{x} \) may be called mirror code for \( x \) and its use, mirror arithmetic, owing to the same just given to \( I \). (Fig. 2). Let us denote the (new) number \( \hat{x} \) with digits \( \hat{x}_i \) in ordinary binary.

2. Relation of Mirror Code and Ordinary Code

From (7) and (11) we deduce that

\[
-\nu(a) 2^n + \nu(x') 2^n + \nu(x'_{n-1}) 2^{n-2} - \nu(x_1) 2^0 = 0
\]

\[
x + \hat{x} = 0
\]

and that digits \( \hat{x}_i \) are those of the additive inverse \( x' \) given by \( 2^n \)

\[
\begin{cases}
\hat{x}_i = x_i \oplus r_i \\
\hat{x}_{n-1} = x_1 \oplus r_{n-1}
\end{cases}
\]

Thus going from \( I \) to \( \hat{I} \) for a given number \( x \), changes its representation \( b \hat{x} \) associated with the additive inverse of \( x' \), also arithmetically obtainable by subtraction from zero or \( 2^n \) (inversion base \( 2^n \)). Zero is unchanged, and non-zero numbers are transformed according to the well-known set of rules (11). These, although required for combinational inverters, (sign-flippers4,9) yet are actually rarely used because generation of the additive inverse is most frequently combined with a simultaneous addition or subtraction, which allows then to dispense with \( \hat{x} \) and use only \( x \) (Boolean bit by bit complement) combined with a least significant weight "not one". This is to say that although the Boolean relation between ordinary code and mirror code may be shown to be a known one (11), the hardware to switch has a non-negligible cost, which, if avoided, implies to choose between the ordinary code or the mirror code. In this respect the situation is different from the distinction between positive and negative logic which amounts only to bitwise reinterpretation of digit values.

We now turn our attention to this last representation, based upon (7) and (8). In the set \( I \), only one power of two is negative, which is the least number allowed to represent signed numbers. However, representation of numbers in signed-digit \( \hat{I} \) is, except for 0, not unique and we may similarly try to use other powers of two to represent signed numbers. Doubling the inventory \( I \), by adjoining to it all powers in it with signs reversed, would be the general signed-digit case. Let us constrain it for economy's sake and use only the set formed of the powers with signs reversed.

\[
I = \{+2^n, -2^{n-1}, \ldots, -2^1, \ldots, -2^0\}
\]
To use one system rather than the other, does not make great practical difference at the start. There is only a minute difference in the range of numbers represented:

\[
x \in [-2^n, 2^n - 1] = D
\]

\[
x \in [-2^n, -2^n + 1] = \hat{D}
\]

\[
D \cap \hat{D} = [-2^n + 1, 2^n - 1]
\]

\[
D \cup \hat{D} = [-2^n, 2^n]
\]

For usual values of \(n\), such differences may be neglected for a choice between the two systems, which then leads to staying with the existing one. These differences will be of interest, only if \(n\) is very small, which give them great relative value, or if for ordinary \(n\), both systems are used in alternation within the arithmetic unit and logic of the computer.

### 1. One Bit Mirror Arithmetic

If \(n\) is made equal to 1, then the domains \(D\) and \(\hat{D}\) become significantly different (Fig. 3).\footnote{\(D = \{-2, -2^0, 0, 2^0\}\) (15) \(\hat{D} = \{-2, 0, 2, 2^0\}\) (17)}

Set \(\hat{D}\) instead of \(D\) will be useful whenever we wish to represent \(+2\) with the same ease as \(-2\), i.e., with two digits only. We now give two examples of application where introducing \(\hat{I} \neq \hat{D}\) will provide an alternate approach to a problem.

#### 3.1 Design of adder

Assume we want to design an adder implementing

\[
x + y - v(r_0)
\]

Where \(x, y\) are \((n+1)\)-digit numbers, \(r\) is 0 or 1 and, contrary to ordinary addition, minus sign precedes \(v(r_0)\). Such adders may be used for instance to implement addition modulo \(2^k + 1\). Then \(x + y - v(r_0)\) may be obtained easily in mirror code, and then reconverted to ordinary code. The mirror code just plays then the role of a useful intermediate representation in the solution process, just as sometimes a change of positive logic to negative logic may be used to afford a convenient reasoning. \(\hat{D}\) however consider first the runs of \(x\) and \(y\) (least significant digits) of \(x\) and \(y\). We must compute in mirror code

\[
x_0 \cdot y_0 \cdot r_0 = v(x_0) + v(y_0) - v(r_0)
\]

\[
\begin{array}{c|c|c|c|c|c|c}
 x_0 & y_0 & r_0 & v(x_0) + v(y_0) - v(r_0) & s_0 & s_1 & s_2 \\
 \hline
 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
 1 & 1 & 0 & 2 & 0 & 0 & 1 \\
 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\[(\text{Table 1:} \quad s_2 s_1 s_0: \text{ordinary code} \quad \hat{s}_2 \hat{s}_1 \hat{s}_0: \text{mirror code})
\]

\[
v(x_0) + v(y_0) - v(r_0) = 2^2 a_2 a_1 a_0 + 2^0 a_0 = 2^1 a_1 + 2^0 a_0
\]

\[
\hat{s}_2 \hat{s}_1 \hat{s}_0 = 0 0 1 \quad \epsilon \{0, 1\}
\]

From Table 1, we deduce

\[
s_0 = x_0 \oplus y_0 \oplus r_0
\]

\[
\hat{s}_2 \hat{s}_1 \hat{s}_0 = \hat{x}_0 \hat{y}_0 \hat{r}_0 + \hat{x}_0 \hat{y}_0 \hat{r}_0
\]

\[
\hat{r}_0 = \text{maj} (x_0, y_0, r_0)
\]

where \(\text{maj}(u, v, w)\) denotes the three-variable majority function. This can be seen as a borrow logic with permuted inputs and combined with a code converter according to (13) yields a possible adder design (Fig. 4). A more classical approach would be to write

\[
x + y - v(r_0) = -[(+) + (-y) + v(r_0)]
\]

which would require three sign-flippers\footnote{\(\frac{x + y - v(r_0) = x - (y) - v(r_0)\) (24)} and an adder. Another would be to write

\[
x + y - v(r_0) = x - (y) - v(r_0)
\]

which would require one subtractor and a sign-flipper on the \(y\) input (Fig. 5) a solution with complexity similar to that of Fig. 4 but different. Finally, a solution with a layout close to the first one (Fig. 4) would be to use an ordinary adder with zero carry brought in \(x + y\) and conditional subtract 1 with a combinational decrementer\footnote{\(x + y - v(r_0) = x - v(r_0)\) (25)}.

### 3.2 \(n+1\)-digit binary arithmetic

Consider two numbers \(x, y\) with

\[
x = \sum_{i=0}^{n} x_i 2^i
\]

\[
y = \sum_{i=0}^{n} y_i 2^i
\]

\[
\epsilon \{0, 1\}
\]

Mirror representation affords a more visual proof of the property\footnote{\(x_0 \cdot y_0 \cdot r_0 \Rightarrow x_0 \cdot y_0 \cdot r_0\)} according to which generalized carry propagation may be restricted to three stages (two interstage intervals) by a suitable coding of interstage carries ("transfers") and use of three layers of cells which extend the usual halfadders\footnote{\(x_0 \cdot y_0 \cdot r_0 \Rightarrow x_0 \cdot y_0 \cdot r_0\)}. To do this, we start from operators with two inputs in \(1, 0, 1\) and propose for an adder made of half-adders only\footnote{\(x_0 \cdot y_0 \cdot r_0 \Rightarrow x_0 \cdot y_0 \cdot r_0\) and Fig. 10 uses this to show that at level 0 the sum \(s_0\) is in \(\{1, 0\}\) which can be coded with a zero most signifi-}

\[\]
cant digit, i.e. carries out of (0) cells are zero. (a) cells generate carries whenever possible, (b) cells only when unavoidable.

Thus, partial sum \(x + y\) and \(a' + b'\) are coded with two digits, in ordinary mirror code, as indicated in Fig. 10 and carries out of (c) cells may be kept equal to zero.

3.3 Adders with base 2 inputs and base (-2) output.

Adders with ordinary base (+2) inputs may also be modified to have an output in signed-digit code or base (-2) code (Fig. 11, 12, 13). Cells are defined arithmetically in the corresponding graphs. In Fig 13, (a) and (d) cells are defined by the following table:

<table>
<thead>
<tr>
<th>Value of ( u + v )</th>
<th>Direct code</th>
<th>Mirror code</th>
<th>Variable mode code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0 00</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>1 1</td>
<td>0 0 01</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
<td>0 1</td>
<td>1 1 01</td>
</tr>
<tr>
<td>1 1</td>
<td>(d)</td>
<td>(d)</td>
<td>1 1 10</td>
</tr>
</tbody>
</table>

3.3 Variable mode arithmetic

Let us explicitly encode the convention (direct, mirror) used to represent the numbers, using an additional binary digit equal to zero for the direct code (D) and 1 for the mirror code (M).

Consider now two numbers in this code and their sum:

\[
\begin{align*}
x &= A x_{n-1} \ldots x_1 x_0 \\
y &= B y_{n-1} \ldots y_1 y_0 \\
x + y &= S a_{n-1} a_{n-2} a_{n-3} \ldots a_0
\end{align*}
\]

where \(A, B, S, x_1, y_1, a_1, a_0, b, s \in \{0, 1\}\) (26)

Addition in this variable mode code, where \(A, B, S\) explicitly represent the code used, requires to determine the rules for obtaining \(a_i\) at each rank \(i\), together with \(S\).

To do this, we consider the sum of digits at rank zero, for which we do not know yet whether to assume a positive or negative carry-in and try to encode it in direct or mirror code. Then we deduce the operation to be performed at a rank \(i\).

At rank \(0\) we compute the sum of digits \(x_1, y_1\), with suitable signs due to \(A, B\) and express it in direct or mirror code. Thus we have

\[
D_0 = (-1)^y(A) v(x_0) + (-1)^y(B) v(y_0) = (-1)^y(A) v(x_0) + (-1)^y(B) v(y_0) = (-1)^y(A) v(x_0) + (-1)^y(B) v(y_0)
\]

and this entails a carry \((-1)^y(A) v(x_0)\) into rank 1, so that generally, for rank \(i\) we must execute:

\[
D_i = (-1)^y(A) v(x_i) + (-1)^y(B) v(y_i) + (-1)^y(T) v(x_{i-1}) + (-1)^y(U) v(y_{i-1})
\]

Values of \(A, B, S, T, U\) and the domains of \(D_0, D_i\) are given in Table 2. Allowed codes for 2-digit \(D_0\) are derived from Table 3 (a-g).

<table>
<thead>
<tr>
<th>ABS</th>
<th>Domain of Do</th>
<th>Domain of Di</th>
<th>Allowed Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0, 1, 2</td>
<td>00 0, 1, 2, 3</td>
<td>D</td>
</tr>
<tr>
<td>001</td>
<td>0, 1, 2</td>
<td>01 0, 1, 2, 3</td>
<td>D</td>
</tr>
<tr>
<td>010</td>
<td>0, 1, 2</td>
<td>10 0, 1, 2, 3</td>
<td>D</td>
</tr>
<tr>
<td>011</td>
<td>1, 0, 1</td>
<td>01 1, 0, 1, 2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>1, 0, 1</td>
<td>10 1, 0, 1, 2</td>
<td>D</td>
</tr>
<tr>
<td>101</td>
<td>1, 0, 1</td>
<td>01 1, 0, 1, 2</td>
<td>M</td>
</tr>
<tr>
<td>110</td>
<td>2, 1, 0</td>
<td>10 2, 1, 0, 1</td>
<td>M</td>
</tr>
<tr>
<td>111</td>
<td>2, 1, 0</td>
<td>11 2, 1, 0, 1</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 2. Ranges and allowed encodings

<table>
<thead>
<tr>
<th>ABS</th>
<th>Domain of Do</th>
<th>Domain of Di</th>
<th>Allowed Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0, 1, 2</td>
<td>00 0, 1, 2, 3</td>
<td>D</td>
</tr>
<tr>
<td>001</td>
<td>0, 1, 2</td>
<td>01 0, 1, 2, 3</td>
<td>D</td>
</tr>
<tr>
<td>010</td>
<td>0, 1, 2</td>
<td>10 0, 1, 2, 3</td>
<td>D</td>
</tr>
<tr>
<td>011</td>
<td>1, 0, 1</td>
<td>01 1, 0, 1, 2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>1, 0, 1</td>
<td>10 1, 0, 1, 2</td>
<td>D</td>
</tr>
<tr>
<td>101</td>
<td>1, 0, 1</td>
<td>01 1, 0, 1, 2</td>
<td>M</td>
</tr>
<tr>
<td>110</td>
<td>2, 1, 0</td>
<td>10 2, 1, 0, 1</td>
<td>M</td>
</tr>
<tr>
<td>111</td>
<td>2, 1, 0</td>
<td>11 2, 1, 0, 1</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 3 (a-g). Direct and mirror codes in \(D_0\) and \(D_i\) number ranges.

The output code then must be chosen so that \(U\) equals \(S\). The second and seventh lines of Table 2 may be deleted as then lead to contradiction between assumed value of \(S\) and allowed code in the last column and we have now the reduced Table 4:

<table>
<thead>
<tr>
<th>A B</th>
<th>Bit operation</th>
<th>Output Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>(v(x_1) + v(y_1) + v(r_1))</td>
<td>D</td>
</tr>
<tr>
<td>0 1</td>
<td>(v(x_1) - v(y_1) - v(r_1))</td>
<td>D</td>
</tr>
<tr>
<td>1 0</td>
<td>(v(x_1) - v(y_1) + v(r_1))</td>
<td>M</td>
</tr>
<tr>
<td>1 1</td>
<td>(-v(x_1) + v(y_1) + v(r_1))</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 4. Bitwise operations.

For the direct code (D), we have to implement ordinary addition sum and carry logic.
1-4 (line 1) or subtraction logic \(^2,4\) (lines 2 and 4). For the mirror mode, we need a logic which is the same as the one studied above in example 3.1 (up to relabeling of variables), in lines 3 and 5. For the last line it can be checked that this is the same as addition logic. The sum digit is the same in all cases. The carry digit then is given by Table 5.

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>maj ((x_1, y_1, r_1))</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>maj ((x_1, y_1, r_1))</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>maj ((x_1, y_1, r_1))</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>maj ((x_1, y_1, r_1))</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5. Carries for variable mode addition.

Thus the output mode is variable if we insist on single-bit carries and there is some choice for it when \(A\) and \(B\) are different. We can use two types of carry only by a suitable choice of \(S\). Then at most one fixed variable will have to be complemented when \(A, B\) vary. There are two such solutions (Table 6).

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>maj ((x_1, y_1, r_1))</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>maj ((x_1, y_1, r_1))</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>maj ((x_1, y_1, r_1))</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>maj ((x_1, y_1, r_1))</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6. Solutions with fixed place complementation.

Thus we have

\[
(1) \quad r_{i+1} = \text{maj} (x_1, y_1, r_i) \\
S = A
\]

\[
(2) \quad r_{i+1} = \text{maj} (x_1, y_1, A \oplus B, r_i) \\
S = B
\]

Addition may be performed with an ordinary adder logic and conditionally complementing \(x\) (bitwise) if mode of \(y\) is kept for the sum and bitwise complementing \(x\) if mode of \(y\) is used for the output. The condition is whether \(A\) and \(B\) are different. The operation on \(r_{i+1}\) if e.g. \(x_1 \oplus (A \oplus B)\) is performed outside the adder stage is similar to condition switching between add and subtract when we perform \(x_1 \oplus (-1)^{y} v(c)\) with an adder: \(y \oplus C\) (or \(C_{y_1} + C_{y_2}\)) has to be performed for all stages. Here the condition variable has to be obtained from \(A\) and \(B\) (one gate). Furthermore, subtraction may now be obtained by simply changing the mode of the subtract end \(y\), which forms its complement on a single bit basis. If conditional add-subtract is desired of the form \(x \oplus (-1)^{y} v(c)\), then the above expressions for \(s_1\) are unchanged, and for the carries we have

\[
\begin{align*}
\text{r}_{i+1} &= \text{maj} (x_1 \oplus A \oplus B \oplus C, y_1, r_i); S = A \\
\text{r}_{i+1} &= \text{maj} (x_1, y_1 \oplus A \oplus B \oplus C, r_i); S = B
\end{align*}
\]

which is similar to ordinary add-subtract, the only difference being that the condition variable is \(A \oplus B \oplus C\) (or its complement) which may be synthesized in one or two gates.

Thus, with variable mode arithmetic, a slightly redundant mode (1-bit) is used. Addition and addition-subtraction may be kept similar in complexity to ordinary conditional add-subtract. Furthermore, the domain of numbers is now totally symmetrical (-2^p, +2^p). Complementation is then a one-bit operation without overflow and the quantity \((-x + y)\) (called sum; in the case of base minus 2,12-15) may be obtained as easily as \(x + y\). The possibility of using cosums instead of sums in addition, multiplication and division, which appeared as a possible advantage of base minus two versus ordinary binary, thus appears to be applicable also to variable mode arithmetic and without the range disymmetry of base (-2) representation.

3.4 Cellular multiplier

In the Pernarowski,\(^2\) cellular multiplier for signed-numbers cells of several types are needed, some of which reduce to ordinary adders, others requiring new logic for the derivation of which a special formalism is introduced. For 1-bit cells the following operators are necessary,\(^4,16\)

\[
\begin{align*}
(a) & \quad v(x_0) + v(y_0) + v(r_0) = 2^2v(x_1) + 2^0v(s_0) \\
(b) & \quad v(x_0) + v(y_0) + v(r_0) = 2^2v(r_1) - 2^0v(s_0) \\
(c) & \quad v(x_0) - v(y_0) - v(r_0) = 2^2v(r_1) + 2^0v(s_0)
\end{align*}
\]

Cells (a) and (c) can be reduced to ordinary add and subtract stages respectively, while (b) is typically a cell as studied in 3.1 and 3.3 above, with output in mirror code.

For 2-bit cells,\(^4,16\) one is an ordinary 2-bit adder, the other a mirror code 2-bit adder and the third may be viewed as a cascade of both types for 1-bit length.

References


ACKNOWLEDGEMENTS

This work was done while the author was with UCLA Computer Science Department as a US-France exchange scientist. The support of National Science Foundation, Washington and of ENSAE, Toulouse, is gratefully acknowledged.
Fig. 1. Coding of x and \( \bar{x} \) using set \([-2^3, 2^2, 2^1, 2^0]\)

Fig. 2. Coding of x: (a) ordinary set \( I \) of primitive powers \( 2^i \)
(b) mirror image set \( \hat{I} \)
\[
I = \{-2^3, 2^2, 2^1, 2^0\}
\hat{I} = \{+2^3, -2^2, -2^1, -2^0\}
\]

Fig. 3. One bit mirror and ordinary codes
Fig. 4. Adder $x + y = v(r_o)$
$x = x_3x_2x_1y_0$
$y = y_3y_2y_1y_0$

Fig. 5. Another design of adder for $x + y = v(r_o)$

$0$-carry

$1$-carry

$\overline{1}$-carry

$\left( s'_i = x'_i + y'_i + t'_i \right)$

$\left( s'_i = x'_i + y'_i + \overline{t'_i} \right)$

Fig. 6. No-carry condition

Fig. 7. 1 and $\overline{1}$ carry conditions
Fig. 6.
Signed digit adder section

Fig. 9.
Verification of $t'_{i-1} = 1 \Rightarrow t''_{i} \in \{0, 1\}$
Fig. 10. Complete graph for 3-level adder

Fig. 11. Base 2 adder with half-adders (2 sections)
Fig. 12. Base 2 adder with signed-digit output

Fig. 13. Adder with base -2 output