Signed-Digit vs. Conventional Intermediate Representations in Two Operand Adders

D. S. Phatak

Electrical Engineering Department
State University of New York, Binghamton, NY 13902–6000

I. Koren

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003

ABSTRACT

In [1] Srinivas and Parhi presented a fast adder based on an intermediate signed-digit representation. In this paper we derive an improved design by (i) utilizing a different procedure to generate the intermediate signed digit result and (ii) optimizing the encoding used to represent the intermediate result.

The analysis leads to a unified framework which reveals a one-to-one correspondence between adders based on intermediate signed-digit representations and conventional ones, i.e., those that do not use interim signed digit results.

I Introduction

A redundant 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 most arithmetic operations [2, 3, 4]. The SD representation also renders most-significant-digit-first schemes feasible and has been used in on–line arithmetic [3] and digit–pipelined schemes [5]. 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 either 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 [6], making it possible to add two numbers in fixed time, irrespective of the word length. This can be of great advantage in multi–operand addition (such as partial product accumulation in a multiplication) or iterative add/subtract operations (like those encountered in division, square root extraction and elementary function evaluation using the CORDIC method). Utilizing an SD representation is analogous to solving the problem in the transform domain. Once in the transform domain, the addition is carry–free and consequently, the time per addition is fixed and independent of the word length. Furthermore, since a bit is just a special case of a signed digit, the conversion from two's complement to SD format is trivial and can be performed in a (small) fixed time irrespective of the word length. The final step, upon the completion of all the operations, is to convert the result back to two's complement format (analogous to an inverse transformation). There has been a resurgence of activity in this area in the recent past, and using the SD representation has been shown to lead to faster implementations for almost all operations.

What is not so obvious is whether going through an intermediate SD representation can be advantageous for the basic two operand addition. Here, it appears that the overhead of converting the signed digit result back to two's complement format is likely to make the implementation at least as slow as (if not slower than) a direct addition. The conversion of a signed–digit number to two's complement format requires a "borrow" propagation across the entire word length. In principle, therefore, both the operations (direct addition of two operands and conversion of a SD number into two's complement) have the same complexity.

In [1] Srinivas and Parhi presented a fast adder based on an intermediate signed-digit representation. What makes this adder have a delay comparable to that of a conventional one is the fact that the carry (or borrow) propagation in both methods is across the entire word length and hence can be expected to be executed in similar time delays. Furthermore, the bit-wise P and G signals required by the conventional adder and the S and Z signals representing intermediate signed digit results require comparable delays to be generated once the operand bits are available.

In this paper we derive an improved adder architecture (based on intermediate signed-digit representation) by
(i) utilizing a different procedure to generate the intermediate signed digit result, and
(ii) optimizing the encoding used to represent the intermediate result.

The analysis shows that no matter how the addition operation is interpreted (as a conventional addition or via an intermediate signed-digit representation), the basic complexity is identical. Thus the signed-digit representation turns out to be just an *encoding* of the intermediate bit-wise result, as much as the conventional P and G signals constitute an *encoding* of the intermediate bit-wise sum. We then show a one-to-one correspondence between adders based on signed-digit representations and conventional ones that do not use intermediate signed digits, which demonstrates that the two methods have the same complexity in principle.

The rest of the paper is organized as follows. The next section briefly describes the conversion algorithm which was independently proposed in [7] and [8] and the "sign–select" implementation of that algorithm originally proposed in [7]. Section III describes our addition algorithm using intermediate signed digits and derives optimal encodings for the intermediate representation. Section IV illustrates a 32 bit adder based on one of the optimal encodings and compares it with that of [1]. Section V establishes the one-to-one correspondence between adders based on intermediate signed-digit representations and conventional ones. The last section presents conclusions.

II Conversion of Signed-Digit Numbers into Two's Complement Format

In many commonly used conversion algorithms [7, 8, 9] the binary signed digits are examined from right to left, i.e., from the least significant digit (LSD) to the most significant digit (MSD) one at a time. Basically there is a need to remove all the occurrences of -1 and "forward" the negative signs all the way to the MSD, which is the only position with negative weight in the two's complement representation. The rightmost -1 is replaced by a 1 and a carry of weight -1 or a "borrow" is forwarded to the next (adjacent left) digit position. This essentially amounts to representing a -1 with $\overline{1}1 = -2+1$, where $\overline{1}$ denotes a -1. The process of forwarding the borrow is continued, replacing 0's by 1's (since $0\cdots0\overline{1} = \overline{1}\cdots11$). If a 1 is encountered, it "consumes the borrow", since $1+(-1)=0$. Finally, if a $\overline{1}$ is encountered during the forwarding of the borrow, it is replaced by 0 and the forwarding continues, or in other words, a $(\overline{1}+\overline{1})$ is replaced by $\overline{1}0 = -2$ as required. Table 1 below summarizes the algorithm converting the binary SD number $y_{n-1}y_{n-2}\cdots y_0$ to its equivalent two's complement representation $d_{n-1}d_{n-2}\cdots d_0$. The variables c_i and c_{i+1} denote the incoming and outgoing borrows, respectively. Note that the borrows always have a negative weight.

y_i	c_i	d_i	c_{i+1}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0
$\overline{1}$	0	1	1
$\overline{1}$	1	0	1

Table 1: Rules for selecting the next borrow c_{i+1} and output bit d_i based on the digit y_i and incoming borrow c_i.

It is easy to verify that the rows of the table satisfy the arithmetic relation

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

In effect, this algorithm can be viewed as the inverse of the canonical recoding algorithm [9], wherein a given two's complement number is recoded so as to have the minimum number of non zero digits. Canonical recoding can be important in a multiply operation where minimizing the number of non zero digits in the multiplier results in the minimum number of add/subtract operations. The canonical recoding algorithm satisfies [9]

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

which is obtained simply by a rearrangement of equation (1) above.

Next, we briefly outline a fast implementation of the outgoing borrow signal c_{i+1} used in the above algorithm [1]. Note that each binary signed digit y_i can have 3 possible values $\{-1, 0, 1\}$ and therefore needs 2 bits to represent. Suppose that the two bits are denoted T_i and R_i, we encode the digit y_i in the following manner:
(i) R_i equals 1 if the digit y_i is 0, and is 0 otherwise,
(ii) T_i indicates the "sign" (or polarity) of the digit y_i: it is 0 if the digit y_i is non–negative (i.e., 0 or 1) and 1 when y_i equals -1. Then, the truth table for c_{i+1} in terms of the input bits c_i, T_i and R_i is as shown in Table 2.

T_i	R_i	c_i	c_{i+1}
0	1	0	0
0	1	1	1
0	0	0	0
0	0	1	0
1	0	0	1
1	0	1	1
1	1	0	$\times$
1	1	1	$\times$

Table 2: Truth table for the outgoing borrow c_{i+1} in terms of the inputs T_i, R_i and c_i. "$\times$" indicates a "don't care".

If both the don't cares in the above table are assigned the value "1" (as is usually done in minimizing logic functions with Karnaugh maps), one obtains the Boolean equation

$$c_{i+1} = T_i + R_i c_i \tag{3}$$

where "+" indicates logical OR and "$\cdot$" (or a product term) indicates logical AND. Comparing the above equation with the conventional fundamental carry propagation Boolean equation

$$c_{i+1} = G_i + P_i c_i \quad \text{where} \quad G_i = a_i \cdot b_i \ \text{ and } \ P_i \in \{(a_i + b_i), (a_i \oplus b_i)\} \tag{4}$$

indicates that T_i is analogous to the G_i or the carry generate signal in conventional addition while R_i is analogous to the P_i or the carry propagate signal. This can be better understood by noting that the information forwarded by the carry (borrow) signal in this case is essentially the negative sign. A borrow is generated when a $\overline{1}$ is encountered and is propagated as long as the current digit (y_i) is zero (in which case R_i is 1). Thus R_i is equivalent to the carry propagate signal in ordinary addition. We therefore refer to R_i as the sign-propagate signal and T_i as the sign-generate signal.

Next, we illustrate a multiplexor based implementation [1] of equation (3). In Table 2 above, notice that if the digit y_i is zero, then the borrow-out (c_{i+1}) equals the borrow-in (c_i). If the digit is non-zero, then the sign of the digit "dominates", i.e., if y_i is positive then c_{i+1} is 0 and if y_i is negative then c_{i+1} is 1. Thus the borrow-out equals the "sign" of y_i when the digit is non-zero. This immediately suggests a MUX based implementation:

$$\begin{aligned} c_{i+1} &= c_i \quad \text{if} \quad R_i = 1 \\ &= T_i \quad \text{if} \quad R_i = 0 \end{aligned} \tag{5}$$

Another way of arriving at this implementation is to assign the Boolean values 0 and 1 to the don't cares in rows 7 and 8, respectively, of Table 2. The resultant Boolean equation is

$$c_{i+1} = \overline{R_i} \cdot T_i + R_i \cdot c_i \tag{6}$$

which can be implemented by a multiplexor (illustrated in Figure 1) which selects the incoming sign or the sign of the current digit depending on whether the digit value is zero or not. Hence the name "sign–select" circuit [1]. It can be seen that the "sign–selection" technique is analogous to the idea of "carry skips" which is utilized in a conventional Manchester adder [9]: if the conventional carry propagate signal P_i is restricted to be the bit-wise exclusive OR ($a_i \oplus b_i$) then the basic carry-propagation equation (4) can also be written as

$$c_{i+1} = P_i \cdot c_i + \overline{P_i} \cdot G_i \tag{7}$$

Since equations (3) and (4) are identical in form, it follows that unrolling the recursion in (3) also leads to equations identical in form to the fundamental carry operation [9, 10, 11], denoted by "$\odot$", and defined by

$$(P, G) \odot (\tilde{P}, \tilde{G}) = (P \cdot \tilde{P}, G + P \cdot \tilde{G}) \tag{8}$$

Let $T_{i:j}$ and $R_{i:j}$ denote the sign-generate and sign-propagate signals for the group of bits in positions $i, i-1, \cdots, j$ where $i > j$. Then, the same fundamental carry operation in equation (8) can be used to generate the group R and T signals for a group of bits $[i : j]$ (with $i > j$) from the R and T signals of two adjacent or overlapping subgroups of bits $[i : m]$ and $[v : j]$ with $i \geq m > j \;\; ; \;\; i > v \geq j$ and $v \geq m - 1$ as follows:

$$\begin{aligned} (R_{i:j}, T_{i:j}) &= (R_{i:m}, T_{i:m}) \odot (R_{v:j}, T_{v:j}) \quad \text{where} \quad i \geq m \;\text{ and }\; v \geq m-1 && (9) \\ c_i &= (R_{i-1:j}, T_{i-1:j}) \odot (1, c_j) && (10) \end{aligned}$$

Equations (9) and (10) are not the only way to express the group R and T signals in terms of those of subgroups. For instance, from equation (6) it is possible to derive different expressions for the group $R_{i:j}$ and $T_{i:j}$ signals in terms of the R and T signals of two adjacent or overlapping subgroups by defining:

$$\begin{aligned}
(R,T) \otimes (\tilde{R},\tilde{T}) &= (R \cdot \tilde{R}, T \cdot \overline{R} + \tilde{T} \cdot R) && (11)\\
(R_{i:j}, T_{i:j}) &= (R_{i:m}, T_{i:m}) \otimes (R_{v:j}, T_{v:j}) \quad \text{where} \quad i \geq m \ \text{ and } \ v \geq m-1 && (12)\\
c_i &= (R_{i-1:j}, T_{i-1:j}) \otimes (1, c_j) && (13)
\end{aligned}$$

In other words, the sign–select operation can be extended to a group of digits. For example, assume that two signals $R_{k:j}$ and $T_{k:j}$ are available for a group of $(k-j+1)$ digits, $k, k-1, \cdots, j$. Group signal $R_{k:j}$ is 1 if each of the digits in the group is zero and is 0 otherwise, while the variable $T_{k:j}$ represents the sign of the most significant non zero digit in the group. The same sign–select circuit shown in Figure 1 can be used to select the outgoing borrow c_{k+1} from the three signals $R_{k:j}, T_{k:j}$ and the borrow into the group, i.e., c_j. If all the digits in the group are zero, then the incoming borrow simply propagates. If not, then the sign of the most significant non zero digit in the group (which is the same as the sign of the whole group) dominates over the incoming borrow. This implies that one can construct a tree of multiplexors for a fast propagation of the borrow. Such a tree based on the efficient sign–select conversion technique was at the heart of the architecture proposed in [1]. The maximum size of a group depends on how many transmission gates can be cascaded in series, in a given technology.

If the conventional bit-wise carry-propagate variable P_i is restricted to be the EXOR of the operand bits ($P_i^{\oplus} = a_i \oplus b_i$), then it can be shown that the pair of variables $(P_i^{\oplus}, Q_i)$ where

$$\begin{aligned}
Q_i &\in \{Q_i^{'}, Q_i^{''}, Q_i^{'''}\} \qquad \text{and} && (14)\\
Q_i^{'} &= a_i \cdot b_i && (15)\\
Q_i^{''} &= a_i && (16)\\
Q_i^{'''} &= b_i && (17)
\end{aligned}$$

also lends itself to a multiplexor-based implementation:

$$\begin{aligned}
(P_{i:j}^{\oplus}, Q_{i:j}) &= (P_{i:m}^{\oplus}, Q_{i:m}) \otimes (P_{v:j}^{\oplus}, Q_{v:j}) \quad \text{where} \quad i \geq m \ \text{ and } \ v \geq m-1 && (18)\\
c_i &= (P_{i-1:j}^{\oplus}, Q_{i-1:j}) \otimes (1, c_j) && (19)
\end{aligned}$$

Relation (18) shows that the $P^{\oplus}$ and Q signals for a group of bits $[i:j]$ (with $i > j$) can be synthesized from the $P^{\oplus}$ and Q signals of overlapping or adjacent subgroups of bits $[i:m]$ and $[v:j]$ by using a multiplexor-based selection circuit.

We conclude this section with a brief description of the method used in [1] to arrive at the intermediate signed-digit result; the encoding used to represent it; and the CMOS cell which is replicated to generate it. Note that the algorithm in Table 1 is independent of the encoding used to represent the signed digits. In [1] the encoding used for a signed digit is $y = y^* - y^{**}$, where $y^*, y^{**} \in \{0, 1\}$. A signed digit is in effect expressed as a difference of two bits. With this encoding, "00" and "11" represent the digit value 0, "10" represents +1 and "01" represents a -1. To generate such a number from the input operands bits a_i and b_i, the arithmetic equation

$$a_i + b_i = 2y^*_{i+1} - y^{**}_i \tag{20}$$

is used, where

$$y^{**}_i = a_i \oplus b_i \tag{21}$$

All the variables in equations (20) and (21) are ordinary bits (i.e., assume one of the two values {0,1}).

Note that the above equation implicitly indicates signal propagation between the neighboring digit positions since the signal y^*_{i+1} for digit position $(i + 1)$ comes from the ith position. Recall that the sign select conversion basically needs the R (zero indicator signal), its complement $\overline{R}$, and T (sign information) bits for each digit (the subscripts in R_i and T_i are omitted whenever a reference to the digit position is not important). The information as coded by digits $y^*_i y^{**}_i$ must therefore be used to generate the signals $R_i, \bar{R}_i$ and T_i. The resulting cell that generates the R and T signals is shown in Figure 2. The circuit which generates the intermediate signed digit result is synthesized by abutting the cells shown in Figure 2, with $y^*_0 = 1$. Once the R and T signals are generated, the sign–select conversion begins with an incoming borrow $c_0 = 0$, which corresponds to setting y^*_0 equal to one [1].

III Improved Procedure to Generate an Intermediate Signed-Digit Result and Derivation of Optimal Encodings

We next demonstrate that adopting a different method to generate the intermediate signed-digit representation and employing different encodings for a signed digit leads to substantial savings in the number of transistors and the critical path delay.

(A) Addition Algorithm : We re-write $(A + B)$ as

$$A + B = (A - \overline{B} - 1) \text{ modulo } 2 \tag{22}$$

where $\overline{B}$ is the one's complement of B:

$$\overline{B} = 2^n - 1 - B \quad \text{, where } n \text{ is the word-length} \tag{23}$$

Note that the one's complement $\overline{B}$ is obtained simply by inverting all the bits of B. The -1 in equation (22) can be taken care of by forcing a carry (borrow)-in $c_0 = -1$. 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 and c_{n-1} represent the borrow-in and borrow-out of the MSB position, and
(ii) a logical "1" is used to represent a borrow of algebraic value -1 (which implies a logical "0" indicates no borrow or a borrow of value 0);
it can be shown that

$$\text{overflow} = c_n \oplus c_{n-1} \quad \text{as in conventional addition.} \tag{24}$$

One is then left with the generation of the signed-digit intermediate result $(A - \overline{B})$, which can be done very fast and in parallel as follows. Note that each of the bits of A and $\overline{B}$ is in the range {0,1}. Hence a bit–wise subtraction directly leads to a signed-digit output representing $(A-\overline{B})$, where each digit is in the range $\{-1, 0, 1\}$. The bit–wise subtraction can be carried out simultaneously (i.e., in parallel) for all the digit positions because there is no need to propagate signals from one digit position to the next.

For the purpose of illustrating our method, consider the case where the two's complement encoding is used to represent a signed digit (we derive optimal encodings a bit later in the section). In this encoding, a signed digit y is represented by two bits $y^s y^a$, where $y^s y^a = 00$ represents the value 0, the bit combination "01" represents the value 1, and "11" represents the value -1. In other words, $y = -2y^s + y^a$, which is a two's complement representation where y^s is the sign bit with negative weight. The pattern "10" is not used at all in this encoding. The advantage of such an encoding is obvious: $\overline{y^a}$ can be utilized in place of the signal R, and T is simply the bit y^s.

Given two numbers $A = (a_{n-1} \cdots b_0)$ and $B = (b_{n-1} \cdots b_0)$ in two's complement format, it is very easy to generate the intermediate signed-digit representation for $(A - \overline{B})$. A cell that accepts bits a_i, b_i and generates a signed-digit output corresponding to the bit–wise subtraction $a_i - \overline{b_i}$ is shown in Figure 3. Note that all the digit positions are independent of each other and there is no signal propagation whatsoever from one digit position to the next. Since both R and $\overline{R}$ are required for the sign–select multiplexing, the cell in Figure 3 shows an XOR–XNOR pair instead of only an XOR gate.

From Figures 2 and 3, the savings in critical path delay as well as transistors are obvious. The cell in Figure 2 needs 26 transistors, while our cell requires only 14 transistors. Note that the cells shown in Figures 2 and 3 are the leaf nodes in the look–ahead tree. This, combined with the fact that the number of leaf nodes in a tree equals (1 + the number of internal nodes) implies that any saving in the transistor count in the leaf cells leads to a substantial saving in the overall transistor count of the adder.

While the two's complement encoding in conjunction with the intermediate signed result generation proposed above leads to a substantial improvement in the delay and transistor count, it is still not optimal, as shown next.

(B) Derivation of Optimal Encodings : Note that the sign selection operation basically needs the "zero" indicator signal R (and its complement $\overline{R}$), and the "sign" indicator T. The actual encoding used to represent a signed digit is of secondary importance, so long as the R and T signals can be generated as fast as possible. In fact, since exactly two bits are required to encode the 3 values $\{-1, 0, 1\}$ that a signed digit can assume, the encoding selected to represent signals R_i and T_i can itself be deemed to be the encoding used to represent a signed digit. Thus, we turn the problem around and select an encoding for R and T which enables their generation in the shortest possible time (and possibly simultaneously with minimum number of transistors). The selected encoding for R and T defines the encoding used to represent a signed digit in the intermediate result.

First, note that the "R_i" signal must be completely specified for all possible input combinations for the MUX implementation of Figure 1 to work properly. In other words, there are no "don't–cares" in the specification of R_i. It has, therefore, only the following two trivial encodings:
(i) $R_i = 0$ when the digit $y_i = a_i - \bar{b}_i = 0$ and $R_i = 1$ otherwise (i.e., when $y_i = \pm 1$).
(ii) $R_i = 1$ when $y_i = 0$ and $R_i = 0$ otherwise.
The truth table for the Boolean variable R_i for each of these two encodings is as shown below.

a_i	b_i	$\bar{b}_i$	$y_i = a_i - \bar{b}_i$	R_i^1	R_i^2
0	0	1	-1	0	1
0	1	0	0	1	0
1	0	1	0	1	0
1	1	0	$+1$	0	1

Table 3: Truth table for two possible encodings R_i^1 and R_i^2 as functions of the input operand bits a_i and b_i

The two encodings are actually equivalent: each requires an XOR–XNOR pair to implement the R, $\overline{R}$ signals and either of them leads to the same critical path delay.

Next we consider encodings for T_i. Note that whenever the digit y_i is zero, the value assigned to sign bit T_i is immaterial because T_i is simply shunted out by the MUX (or in other words, the input sign propagates). This introduces "don't–cares" in the truth table for T_i. The value of T_i matters only when the signed digit y_i is non zero. Once again, there are two possible encodings for T_i,
(i) $T_i = 1$ when $y_i = a_i - \bar{b}_i = 1$; $T_i = 0$ when $y_i = -1$ and $T_i = \times$, i.e., a "don't–care" when $y_i = 0$.

(ii) $T_i = 0$ when $y_i = a_i - \bar{b}_i = 1$; $T_i = 1$ when $y_i = -1$ and $T_i = \times$ when $y_i = 0$.
The truth table for these encodings is shown below in Table 4.

a_i	b_i	$\bar{b}_i$	$y_i = a_i - \bar{b}_i$	T_i^1	T_i^2
0	0	1	-1	0	1
0	1	0	0	$\times$	$\times$
1	0	1	0	$\times$	$\times$
1	1	0	$+1$	1	0

Table 4: Truth table for two possible encodings T_i^1 and T_i^2 as functions of the input operand bits a_i and b_i

Each of the don't cares in the 5th column of the above table can take one of the two values (0 or 1). Thus there are 4 different sub encodings corresponding to the 4 possible assignments of binary values to the don't cares in column 5 (these will be denoted as T^{10}, T^{11}, T^{12} and T^{13}, corresponding to the don't care assignments 00, 01, 10 and 11, respectively). Similarly, there are 4 possible sub-encodings generated by column 6 (which will be denoted by T^{20}, T^{21}, T^{22} and T^{23}). Table 5 below shows the logical equation for T_i as a function of a_i and b_i for each of these 8 distinct encodings. The transistor count associated with each of the encodings is also listed in the table.

Encoding	Logical Function for T_i	Transistor Count	Comments (Gates required)
T^{10}	$a_i \cdot b_i$	6	NAND + inverter
T^{11}	a_i	0	a_i itself serves as signal T_i
T^{12}	b_i	0	b_i itself serves as signal T_i
T^{13}	$a_i + b_i$	6	NOR + inverter
T^{20}	$\overline{a_i + b_i}$	4	NOR
T^{21}	$\bar{b}_i$	2	inverter
T^{22}	$\bar{a}_i$	2	inverter
T^{23}	$\overline{a_i \cdot b_i}$	4	NAND

Table 5: The logical functions and corresponding critical path delay and transistor count for each of the eight distinct encodings for signal T_i.

From the table it is evident that encodings T^{11} and T^{12} are the best since these do not require any additional processing: one of the two inputs a_i or b_i itself serves as the "sign" bit of the corresponding signed digit (in the intermediate signed-digit representation). Why can T_i be simplified to this extent ? The answer can be found from Tables 3 and 4. The key is to realize that R_i, being an XOR (or XNOR) of a_i and b_i already has the information on whether a_i and b_i have the same values or different values. Also, from the 4th column of Table 3 (and

Table 4), it is clear that the digit y_i is zero only when a_i and b_i have different values and 1 when a_i and b_i have the same value. Putting these two facts together, each of the two pairs (R_i, a_i) and (R_i, b_i) contains all the information necessary. Thus T_i can be replaced by either a_i or b_i. We would like to point out that an exhaustive listing of all possible encodings for T is done here only for the sake of clarity. The fact that T^{11} and T^{12} are optimal can be directly determined from a Karnaugh map for the 5th and 6th columns in Table 4.

Finally, note that each of the two encodings for R listed in Table 3 can be utilized with each of the two "best" encodings for T, yielding 4 distinct encodings. Each of these encodings leads to the same critical path delay, and the minimum transistor count associated with the XOR–XNOR pair required to generate $(R, \overline{R})$ bits. The encodings are listed in Table 6.

Encoding	Digit values and the (T_i, R_i) pairs used to represent them
(T^{11}, R^1)	$-1 \leftrightarrow (0,0)$ $0 \leftrightarrow$ (0,1) or (1,1) $+1 \leftrightarrow (1,0)$
(T^{11}, R^2)	$-1 \leftrightarrow (0,1)$ $0 \leftrightarrow$ (0,0) or (1,0) $+1 \leftrightarrow (1,1)$
(T^{12}, R^1)	$-1 \leftrightarrow (1,0)$ $0 \leftrightarrow$ (0,1) or (1,1) $+1 \leftrightarrow (0,0)$
(T^{12}, R^2)	$-1 \leftrightarrow (1,1)$ $0 \leftrightarrow$ (0,0) or (1,0) $+1 \leftrightarrow (0,1)$

Table 6: The four optimal encodings which lead to the minimal transistor count.

Table 6 above indicates that the optimal encodings no longer have a unique representation for the digit value $y_i = 0$. In fact it can be easily verified that out of the eight possible encodings for T, the 4 encodings T^{10}, T^{13}, T^{20} and T^{23} lead to a unique representation for the digit value 0. The remaining four encodings, i.e., T^{11}, T^{12}, T^{21} and T^{22} lead to two different bit pairs to represent the digit value 0.

When two different bit pairs are used to represent the same digit value, we have transcended the notion of "intermediate signed-digit representation". In fact it is not discernible that we utilized an intermediate signed-digit representation, because this implies a unique representation for each digit value. What we have done is completely bypassed the "intermediate–signed–digit" notion in its strict sense and directly synthesized the T_i and R_i signals that are required for the carry propagation.

Table 6 shows that in encoding T^{11} a digit with value -1 is represented by a "sign" bit with a value 0. This bit is one of the inputs to the MUX in the sign-selection circuit (as seen in Figure 1). Hence, in the case when the sign "dominates" and gets selected by a multiplexor, a logical (Boolean) value of 0 will appear at the output of the MUX, but the correct algebraic value to be forwarded is -1. In other words, for the simple MUX selection circuit to work along with encoding T^{11}, $c_i = 0$ must represent a borrow of -1 while $c_i = 1$ should correspond to a borrow of value 0 (i.e., no borrow). This means that the encoding for the borrows is the complement of that used in Table 1. Such an encoding of the borrows c_i poses no problem: the borrow propagation operation defined by equation (6) does not impose any particular encoding on variables c_{i+1}, c_i and T_i, *as long as the same encoding is used to represent all of them.* In other words, if $T_i = 0$ corresponds to a sign of -1 then $c_i = 0$ must also represent a borrow of value -1, in which case $T_i = 1$ and $c_i = 1$ represent no borrow or a borrow of value 0. The only other choice is to let $T_i = 0$ and $c_i = 0$ represent an algebraic value of 0 in which case $T_i = 1$ and $c_i = 1$ represent algebraic value -1 (this representation is implied by selection encoding T^{12}). Thus selection of an encoding for the "sign-bit" also determines the encoding of the "borrow" signals.

In contrast, the fundamental carry operation implies that a carry (borrow) of weight $+1$ (-1) *is represented or encoded by a 1*, i.e., it presupposes a fixed encoding.

Since all the four encodings listed in Table 6 are equivalent, we arbitrarily select T^{12} in order to illustrate our adder architecture. Henceforth, all the cell and block diagrams are based on encoding (T^{12}, R^1) in row 3 of Table 6 (For the sake of illustration, we arbitrarily select encoding R^1 for variable R. Thus, the R encoding is implied and need not be stated explicitly while referring to an overall encoding. A cell that implements the R_i and T_i signals is illustrated in Figure 4. Compared to the cell in Figure 2, our cell has a smaller critical path delay, and needs only 10 transistors to implement the XOR–XNOR pair. To reduce the delay, the XOR and XNOR gates are implemented separately, each with a 4 transistor transmission structure [12]. The inversion of one of the signals needs 2 more transistors, for a total of 10 transistors.

IV Illustrative Architecture and Comparison

(A) Architecture The overall architecture of a 32 bit adder based on the T^{12} encoding is shown in Figure 5. It is modular and can be easily extended to word lengths of 64 bits or longer. A 32 bit word length has been selected merely for the convenience of illustration.

As shown in sections II and III, there are several different ways to express the basic carry (borrow) propagation equation. Consequently, several different architectures are possible: for instance, architectures that utilize a combination of two or more of carry (borrow)-skip; carry (borrow)-look-ahead; carry (borrow)-select and a variety of other well-known techniques. We have selected a combination of carry-look-ahead and carry-select techniques for the ease of illustration and comparison with the design proposed in [1], which also employed a combination of the carry look-ahead and carry-select techniques.

The input bits are grouped into blocks, and a ripple-carry addition (**RCA** in Figure 5) is performed twice for each block, with the incoming carry assuming the values 0 and 1. When the actual carry input to a block is generated by the look-ahead tree, it is used to drive a block multiplexor (**BM** in Figure 5) which then selects the correct output.

The larger the block size, the fewer the number of carries to be generated, but the larger the time delay incurred by the ripple-carry addition through the block. The size of the block should therefore be selected in such a way that the delay of the ripple-carry addition through the most significant block and the delay associated with the generation of the carry input to that block are equal or as balanced as possible, so that all the inputs to the block multiplexor arrive at approximately the same time. Table 7 shows the optimal block size b as a function of operand word length, under the assumptions that the block sizes of all bit blocks are equal, and fan-out loads and wire delays are negligible.

Word Length	Ripple-carry block size b
8	2
16	2
32	4
64	4
128	4
256	8

Table 7: Optimal ripple-carry block size b as a function of word length of the input operands, ignoring fanout loads, wire delays, etc.

Selection of the fan-in factor of the look-ahead tree is totally independent of the choice of the block size b for the ripple-carry addition. For instance, the optimal block size changes along with the word length, as shown in Table 1. The fan-in factor of the look-ahead tree, on the other hand, remains two for all these word lengths.

Note that the group of the four least significant bits employs a different circuit labeled **CG** (Carry Generator) in place of the **SSC** circuit. This is done in order to accommodate a variable external carry-in signal, with a negligible or no increase in the total critical path delay. A variable carry-in may be required when the same hardware is used to perform an algebraic subtraction $A - B$, or in a multi-operand addition.

To understand the operation of the **CG** block, note that in Figure 6 if the signal T_0 is replaced by the actual carry c_1 then multiplexor M1 generates

$$\overline{R_1} \cdot T_1 + R_1 c_1 = c_2 \tag{25}$$

i.e., the true carry c_2. This, in turn, causes multiplexor M3 to generate

$$\overline{R_{3:2}} \cdot T_{3:2} + R_{3:2} \cdot c_2 = c_4 \tag{26}$$

i.e., the true carry c_4 as required. Note that the bit-wise propagate signals R_i and $\overline{R_i}$ are the XOR/XNOR functions. The propagation of signals in the look-ahead tree begins after these signals become available. Therefore, if the carry c_1 (or its complement) can be generated within a delay smaller than that required to realize the XOR/XNOR function pair, the total critical path delay remains unaffected. It turns out that $\bar{c}_1$ can be implemented with a delay comparable to that required to generate the XOR/XNOR pair by employing the inverting majority gate (a complex gate) illustrated in Figure 9 at the top.

(B) Comparison

This subsection compares the proposed architecture with that of [1]. Table 8 shows the transistor counts for different word lengths.

Word Length (W)	Our Adder		Adder in [1]
	Allowing up to 4 series transmission gates	Allowing up to 6 series transmission gates	
8	350	258	284
16	796	588	756
32	1636	1276	1772
64	3476	2732	3620

Table 8: Transistor counts for different word lengths. The counts for designs in [1] are reproduced from Table III therein for the sake of comparison. The design in [1] has up to 11 transmission gates in series.

Note that our adder needs a smaller number of transistors than that required by the adder in [1] for word lengths above 32 bits. While the proposed design limits the number of transmission gates in series to 4, the adder in [1] has up to 11 transmission gates cascaded in series, which would make the cascaded chain far slower (than a linear increase). Even when we limit the number of series transmission gates to 4, our 32 and 64 bit adders use fewer transistors.

If we limit the number of series transmission gates to 6 (which is reasonable, especially as the feature size continues to shrink) the improvements in transistor savings are substantial: 28% and 25% respectively, for 32 and 64 bit adders. Thus our adders require considerably fewer transistors. The reduction comes about mainly because of the simplification of the cell that generates the bit-wise T_i and $R_i, \overline{R}_i$ signals. As mentioned before, these cells are the leaf nodes of the look-ahead tree. Since the number of leaf nodes of a tree is (1 + number of internal nodes), transistor savings in the leaf cell lead to substantial reduction in the transistors required in the overall adder.

The addition times for different word lengths are shown in Table 9.

For the sake of consistency the same delay model used in [1] is adopted to estimate the delays of our adders. This model assumes that the delay of XOR/XNOR and two input gates is 1 unit, and ignores fanout loads, wire delays etc. This delay model is not accurate and is used here only for the sake of consistent comparison with the design in [1].

It is clear that our adder is faster than that in [1] for all word lengths. For 32 and 64 bit operands, the speed improvements are 33% and 31%, respectively.

Our adder is always faster than that in [1] because
(i) Our method of intermediate signed digit generation is more straightforward and faster,

Parameters		Addition Time Delays	
Word Length (W)	Our Block size b	Our Adder	Adder in [1]
8	4	6	8
16	4	7	8
32	8	8	12
64	8	9	13

Table 9: Comparison of addition times (gate delays) as functions of word length. Delays for the adders in [1] are reproduced from Table II in that paper for the purpose of comparison. Encoding $S^{\{21\}}$ in Table 6 is used in our designs. For the sake of consistency, the same delay model used in [1] is adopted here.

(ii) We have optimized the encoding used to represent the intermediate signed-digit result, and (iii) Our tree design is faster: in [1], the block size used is 8. A ripple through 8 bits is too slow compared with the propagation of carry signals in our tree which leads us to use a block size of 4. In their design for 32 bit word length, the block ripple is *slower* than the borrow signal generation and it determines the critical path which indicates unbalanced paths.

In summary our design is significantly better than that of [1] in terms of both speed and area. However, the main focus of this paper is not simply a comparison with the design in [1], but to show an equivalence between designs based on intermediate signed digit results and conventional designs which is established in the next section. Given this equivalence, it is clear that going to an intermediate signed digit representation does not yield any advantage over conventional designs that do not go through intermediate signed digits. Thus, it is feasible (and likely) that recent adder designs published in the literature (for instance [13, 14, 15, 16]) are significantly better than designs based on intermediate signed-digit representations.

V Correspondence between Adders Based on Intermediate Signed Digit Representations and Conventional Ones

In the following we demonstrate a one-to-one correspondence between adders employing intermediate signed-digit representations and those based on conventional schemes. This in turn implies that there is no advantage gained by using intermediate signed-digit representation. Let $A = (a_{n-1} \cdots a_0)$ and $B = (b_{n-1} \cdots b_0)$ be the two operands in two's complement format that are to be added to generate a result in two's complement. Next, we use a two-column format to illustrate a side-by-side comparison of the two addition schemes (one based on intermediate SD representation versus conventional addition). This comparison helps bring out the equivalence between the two schemes.

Adders based on intermediate Signed-Digit interpretation	**Conventional Addition**
$A+B$ is implemented as $A - \overline{B} = A + B + \text{ulp} - 2^n$; where $\overline{B} = 2^n - \text{ulp} - B$ is the one's complement of B, obtained by inverting each bit b_i. The extra 1 can be removed by forcing a carry-in of -1 while the -2^n term manifests itself as a borrow-out that can be discarded (or handled properly to detect overflow, etc.)	No special interpretation is used.
Intermediate **bit-wise subtraction** output $y_i = a_i - \bar{bi} \in \{-1, 0, 1\}$.	The **bit-wise algebraic sum** output $y_i = a_i + b_i \in \{0, 1, 2\}$.
The bit-wise difference y_i needs at least two variables to "encode". Let the variable pair used to encode y_i be (T_i, R_i).	Even in conventional addition the bit-wise sum y_i needs two bits to encode. Traditionally, two-variable pairs such as (G_i, P_i) where $P_i = a_i \wedge b_i$ or $P_i = a_i \oplus b_i$, have been used to encode y_i. (G_i, P_i) can be replaced by $(Q_i, P_i^{\oplus})$.
A "borrow" c_i is propagated from LSB to MSB. The algebraic value of c_0 is -1. Borrow c_i into position i has a negative weight, i.e., $c_i \in \{-1, 0\}$ for $i \geq 0$	A "carry" c_i is propagated from LSB to MSB. The algebraic value of c_0 is 0 Carry c_i into position i has a positive weight, i.e., $c_i \in \{0, 1\}$ for $i \geq 0$
Final operation is $w_i = y_i - c_i$ which can result in any of four values $\{-2, -1, 0, 1\}$.	Final operation is $w_i = y_i + c_i$ which can result in any of four values $\{0, 1, 2, 3\}$.
w_i is recoded with two bits: (1) c_{i+1} representing the "outgoing borrow", and (2) the final output bit d_i that satisfy the relation $w_i = y_i - c_i = d_i - 2c_{i+1}$ (27)	w_i is recoded with two bits: (1) c_{i+1} representing the "outgoing carry", and (2) the final output bit d_i that satisfy the relation $w_i = y_i + c_i = d_i + 2c_{i+1}$ (28)

The encoding of bits d_i is fixed because the final sum output must be in two's complement format. Note that the above equations (27) and (28) uniquely determine *the algebraic values of* c_{i+1} and d_i for each possible value of w_i (abbreviated $w_i \leftrightarrow (c_{i+1}, d_i)$) as indicated below:

$-2 \leftrightarrow (-1,0) \quad -1 \leftrightarrow (-1,1)$
$0 \leftrightarrow (0,0) \quad 1 \leftrightarrow (0,1)$

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

The only difference that can arise between the two schemes is therefore due to the encoding used to represent intermediate bit-wise result y_i and the resultant Boolean recursion equations.

There are 24 possible encodings to represent $y_i \in \{-1, 0, 1\}$ with 2 bits $(4 \times 3 \times 2)$

There are 24 possible encodings to represent $y_i \in \{0, 1, 2\}$ with 2 bits $(4 \times 3 \times 2)$

For every encoding (and consequently for every recursion) in one scheme, there is a corresponding encoding (and a corresponding recursion) in the other scheme. This correspondence is summarized in Table 10.

Encodings for bit-wise difference Yi (bits Ti, Ri) in a method based on intermediate SD rep.			Next carry/borrow Ci+1	Sum output di	Correspondence
--1	0	+1			
0 0	0 1	1 0	m(0, 4, 5) + d(3, 7)	m(0, 2, 5) + d(3, 7)	E0 <----> G14
0 0	0 1	1 1	m(0, 4, 5) + d(2, 6)	m(0, 3, 5) + d(2, 6)	E1 <----> G20
0 0	1 0	0 1	m(0, 4, 6) + d(3, 7)	m(0, 1, 6) + d(3, 7)	E2 <----> G8
0 0	1 0	1 1	m(0, 4, 6) + d(1, 5)	m(0, 3, 6) + d(1, 5)	E3 <----> G22
0 0	1 1	0 1	m(0, 4, 7) + d(2, 6)	m(0, 1, 7) + d(2, 6)	E4 <----> G10
0 0	1 1	1 0	m(0, 4, 7) + d(1, 5)	m(0, 2, 7) + d(1, 5)	E5 <----> G16
0 1	0 0	1 0	m(1, 4, 5) + d(3, 7)	m(1, 2, 4) + d(3, 7)	E6 <----> G12
0 1	0 0	1 1	m(1, 4, 5) + d(2, 6)	m(1, 3, 4) + d(2, 6)	E7 <----> G18
0 1	1 0	0 0	m(1, 5, 6) + d(3, 7)	m(0, 1, 6) + d(3, 7)	E8 <----> G2
0 1	1 0	1 1	m(1, 5, 6) + d(0, 4)	m(1, 3, 6) + d(0, 4)	E9 <----> G23
0 1	1 1	0 0	m(1, 5, 7) + d(2, 6)	m(0, 1, 7) + d(2, 6)	E10 <----> G4
0 1	1 1	1 0	m(1, 5, 7) + d(0, 4)	m(1, 2, 7) + d(0, 4)	E11 <----> G17
1 0	0 0	0 1	m(2, 4, 6) + d(3, 7)	m(1, 2, 4) + d(3, 7)	E12 <----> G6
1 0	0 0	1 1	m(2, 4, 6) + d(1, 5)	m(2, 3, 4) + d(1, 5)	E13 <----> G19
1 0	0 1	0 0	m(2, 5, 6) + d(3, 7)	m(0, 2, 5) + d(3, 7)	E14 <----> G0
1 0	0 1	1 1	m(2, 5, 6) + d(0, 4)	m(2, 3, 5) + d(0, 4)	E15 <----> G21
1 0	1 1	0 0	m(2, 6, 7) + d(1, 5)	m(0, 2, 7) + d(1, 5)	E16 <----> G5
1 0	1 1	0 1	m(2, 6, 7) + d(0, 4)	m(1, 2, 7) + d(0, 4)	E17 <----> G11
1 1	0 0	0 1	m(3, 4, 7) + d(2, 6)	m(1, 3, 4) + d(2, 6)	E18 <----> G7
1 1	0 0	1 0	m(3, 4, 7) + d(1, 5)	m(2, 3, 4) + d(1, 5)	E19 <----> G13
1 1	0 1	0 0	m(3, 5, 7) + d(2, 6)	m(0, 3, 5) + d(2, 6)	E20 <----> G1
1 1	0 1	1 0	m(3, 5, 7) + d(0, 4)	m(2, 3, 5) + d(0, 4)	E21 <----> G15
1 1	1 0	0 0	m(3, 6, 7) + d(1, 5)	m(0, 3, 6) + d(1, 5)	E22 <----> G3
1 1	1 0	0 1	m(3, 6, 7) + d(0, 4)	m(1, 3, 6) + d(0, 4)	E23 <----> G9
0	1	2			
Encodings for bit-wise sum Yi in normal addition					

Table 10 : One-to-one correspondence between adders based on intermediate signed digits and conventional ones. Refer to the text for further explanation of the table.

The first column in Table 10 indicates the 24 possible encodings that can be used to represent the 3 intermediate digit-wise results, i.e., $\{-1, 0, 1\}$ in adders based on intermediate signed digits; and $\{0, 1, 2\}$ in conventional adders. The 3 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 3 sub columns 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 bits $(T_i, R_i) =$ (0,0) represents an 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 beginning with “E”, while encodings for conventional adders have labels beginning with “G”. Thus the first row and first column of the table specifies encoding E0 for adders based on signed digits and encoding G0 for conventional adders.

The second column shows the Sum-of-Products (SOP) Boolean expression for the outgoing borrow c_{i+1} in terms of the bits encoding y_i and the borrow-in c_i for *adders based on intermediate signed digits*, i.e., for the “E” encoding under consideration. (i.e., this column gives $c_{i+1} = f(c_i, T_i, R_i)$ in the standard SOP notation including the minterms and the don't cares. Here, T_i, R_i are the bits encoding the digit-wise intermediate result $y_i \in \{-1, 0, +1\}$, and c_i is the borrow-in).

Likewise, the third column gives SOP expressions for the final sum output digit d_i *for the* “E” *encoding under consideration.* The don't cares (in columns 2 and 3) arise because only 3 out of the 4 combinations of two bits are needed to encode all possible values of y_i. The last column indicates the one-to-one correspondence between the two schemes.

The first correspondence indicates that E0 and G14 are equivalent. Encoding G14 corresponds to the 15th row of the table (the rows are labeled starting 0 for convenience), showing that in this encoding, the intermediate bit-wise sum value $y_i = 0$ is encoded by $(T_i, R_i) =$ (1,0); $y_i = 1$ is encoded by “01” while $y_i = 2$ is encoded by “00”.

The equivalence holds in the following sense: under encoding G14 for conventional addition,

$$c_{i+1} = \sum[m(0, 4, 5) + d(3, 7)] \tag{29}$$

$$d_i = \sum[m(1, 4, 6) + d(3, 7)] \tag{30}$$

In other words,
(i) the SOP expression for c_{i+1} is identical under encoding E0 for adders based on signed digits and encoding G14 for conventional adders; while
(ii) the expression for d_i under encoding E0 is the complement of the expression for d_i under G14.

The encodings turn out to be symmetric in the sense that E$i \longleftrightarrow$ Gj and E$j \longleftrightarrow$ Gi for $0 \leq i, j \leq 23$.

In the notation of Table 10 the encoding T^{12} underlying the architecture illustrated in Figure 5 can be shown to correspond to either E14 or E16 (*along with the implicit assignments to the don't cares which can make either* E14 *or* E16 *equivalent to* T^{12}).

It can be verified that several of the encodings in Table 10 lead to look-ahead recursions which can be implemented via multiplexors, so that fast, MUX-based look-ahead trees or carry/borrow-skip circuits can be implemented.

V Conclusion

We have analyzed the design of adders based on intermediate signed-digit representation. The analysis reveals a one-to-one correspondence between these adders (based on intermediate signed digit results) and conventional adders. This implies that the complexity of the two designs (those based on intermediate signed digit results and conventional ones) is identical, which is intuitive: as long the underlying operation is a two-operand addition, whichever way we approach it, the fundamental complexity should remain the same. Thus, there is no advantage gained by adopting an intermediate signed-digit representation (over and above a conventional adder design). Because of this equivalence, Doran's framework for lookahead recursions [17], originally developed for conventional adders, is also applicable to adders based on intermediate signed digits.

References

[1] 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.

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

[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] J. Duprat and J. Muller, "The CORDIC algorithm: new results for fast VLSI implementation," *IEEE Trans. on Computers*, vol. TC–42, pp. 920–930, Feb. 1993.

[5] 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.

[6] 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.

[7] H. R. Srinivas and K. K. Parhi, "High–Speed VLSI arithmetic processor architectures using Hybrid Number Representation," *Journal of VLSI signal processing*, vol. 4, pp. 177–198, 1992.

[8] 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.

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

[10] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. on Computers*, vol. TC-31, pp. 260–264, Mar. 1982.

[11] T. Lynch and E. E. Swartzlander, "A Spanning Tree Carry Lookahead Adder," *IEEE Trans. on Computers*, vol. 41, pp. 931–939, Aug. 1992.

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

[13] 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*, vol. 27, pp. 657–659, April 1992.

[14] V. Kantabutra, "A recursive carry–look–ahead/carry–select hybrid adder," *IEEE Trans. on Computers*, vol. 42, pp. 1495–1499, Dec. 1993.

[15] 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*, vol. 28, pp. 1145–1150, Nov. 1993.

[16] N. Ohkubo and Suzuki, M., et. al., "A 4.4-ns CMOS 54 × 54-b Multiplier Using Pass-Transistor Multiplexor," *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 251–256, Mar. 1995.

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

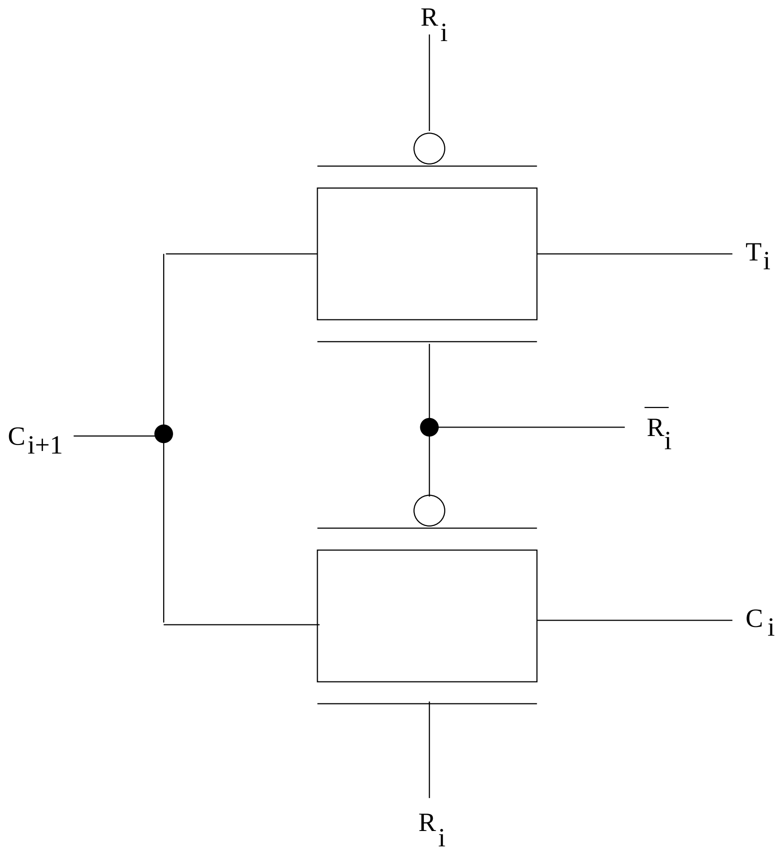


Figure 1: Transmission gate based multiplexor implementation of the sign-selection operation in equation 6

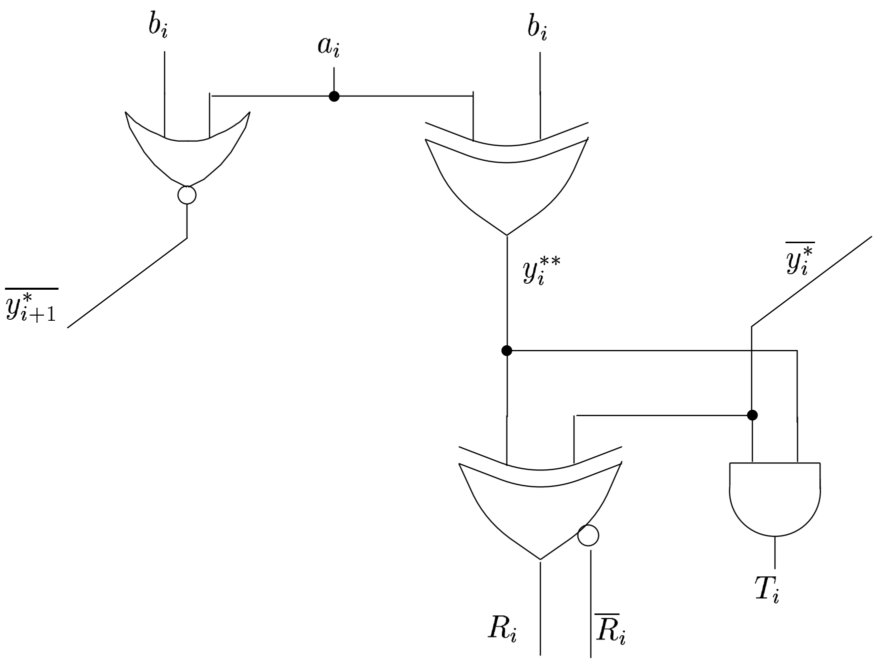


Figure 2 : Cell used in [1] to generate the intermediate signed digit result. A binary signed digit $y \in \{-1, 0, +1\}$ is expressed as a difference of two bits: $y = y^* - y^{**}$, where $y^*, y^{**} \in \{0, 1\}$. For each digit position i (where $0 \leq i \leq (\text{wordlength} - 1)$); signals y_i^* and y_i^{**} are generated as per equations (20) and (21). Note that there is implicit signal propagation between adjacent digits: $\overline{y_i^*}$ (which is the complement of y_i^*) is generated from operands in the $(i-1)$th position. The two gates at the bottom encode the result of subtraction $y_i^* - y_i^{**}$ with S_i and Z_i signals.

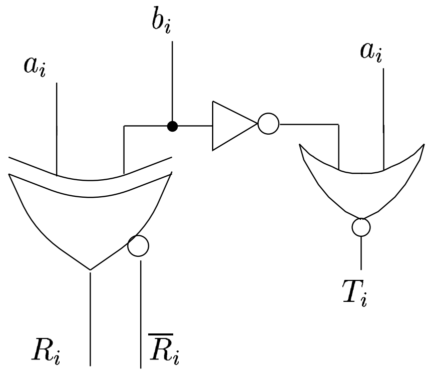


Figure 3 : Cell required to generate the intermediate signed digit result in our method, if straight two's complement encoding (i.e., "00" represents 0, "01" represents +1 and "11" represents −1) is used to represent the intermediate signed digit result. Note that there is no signal propagation between adjacent digit positions. The signals for each position are generated using only the input operand bits in that position. This cell can be further simplified to that shown in Figure 4 by employing the optimal encodings derived in Section III.

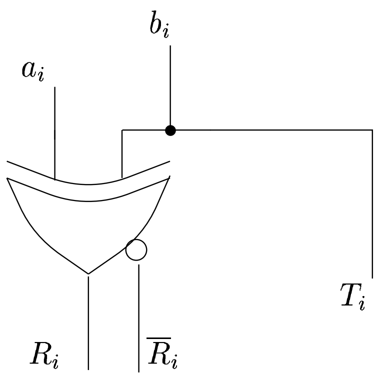


Figure 4: Cell based on encoding (T^{12}, R^1) in Table 6. Note that operand bit b_i itself serves as the "sign-bit" T_i. The simplification over the cell shown in Figure 3 is the result of adopting encoding T^{12} for the sign-bit (instead of the straight two's complement representation on which the cell in Figure 3 is based). This cell has a significantly smaller number of transistors as well as a smaller critical path delay compared to the cell in Figure 2, which was used in [1].

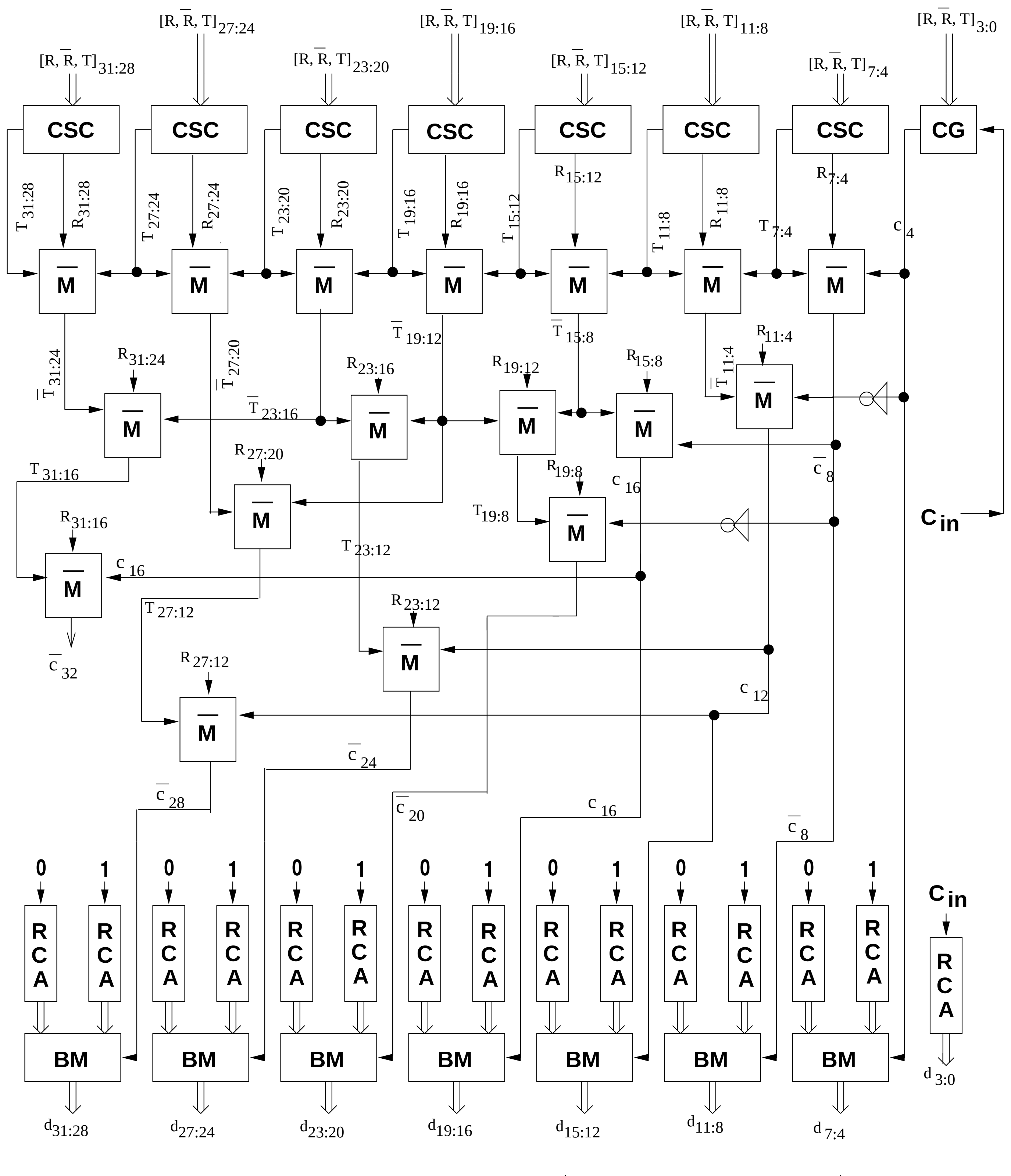


Figure 5 : Overall architecture of our 32 bit adder (refer to the text for details). **SSC** is the sign-select circuit shown in Figure 6, $\overline{\mathbf{M}}$ is the inverting multiplexor shown in Figure 7 (the architecture is flexible and the block $\overline{\mathbf{M}}$ can be replaced with the transmission gate MUX in Figure 1 with minor modifications). **BM** is a 4-bit block multiplexor, and **RCA** is the ripple carry adder shown in Figure 8. **CG** block shown in Figure 9 is used instead of SSC block for the group of 4 least significant digits in order to reduce the critical path delay. Every group $R_{i:j}$ is generated from R signals of the same sub-groups that are combined to generate the corresponding $T_{i:j}$. For example, $T_{27:12}$ is generated from $\overline{T}_{27:20}$ and $\overline{T}_{19:12}$, likewise, $R_{27:12}$ is generated from $R_{27:20}$ and $R_{19:12}$.

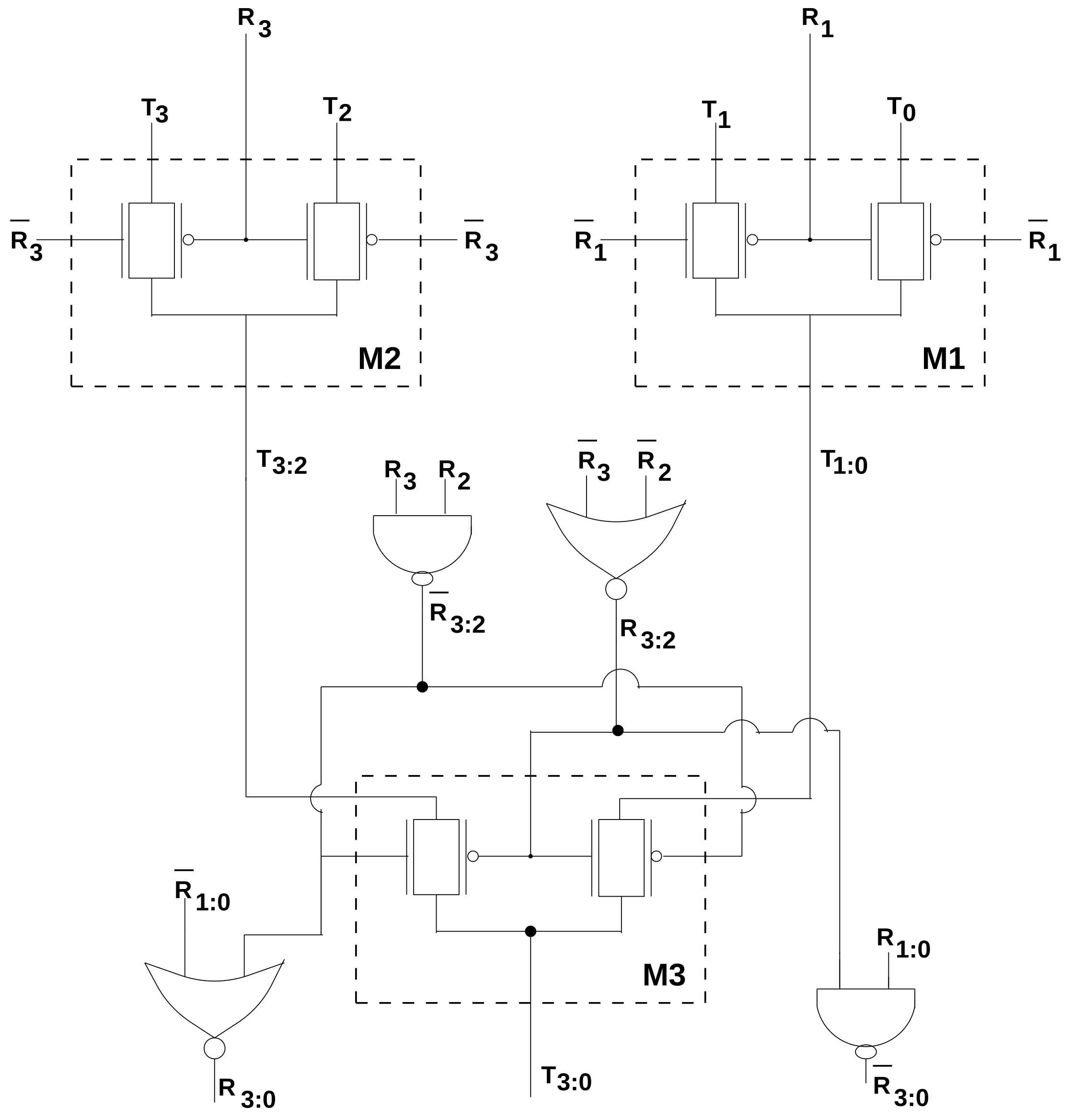


Figure 6 : Sign Select Circuit (SSC) implementing the first two levels of the binary look-ahead tree for sign selection.

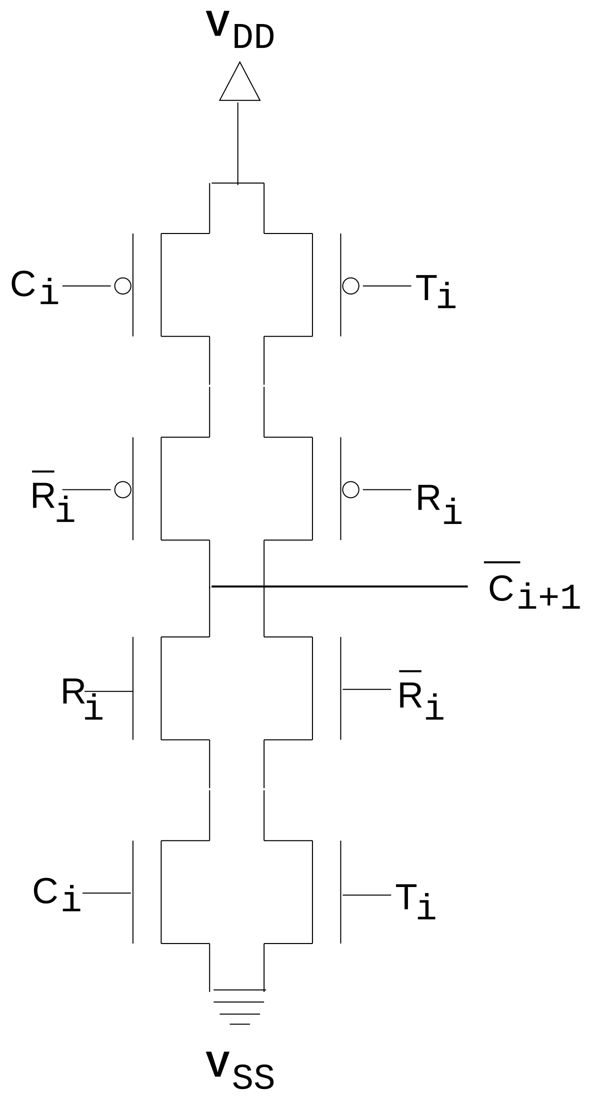


Figure 7 : Alternate design: inverting multiplexor which actively restores the signals.

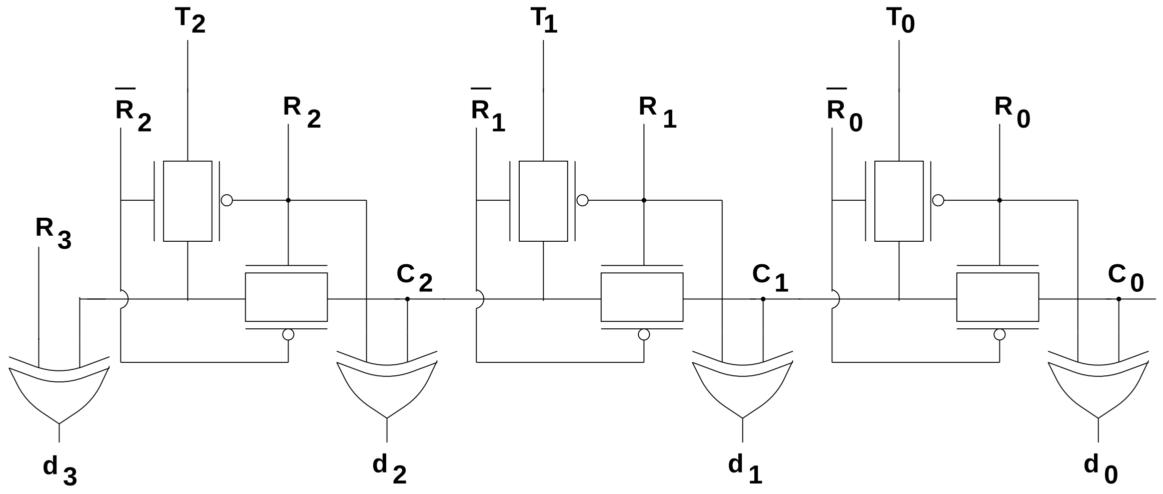


Figure 8 : The ripple-carry circuit for block size $b = 4$.

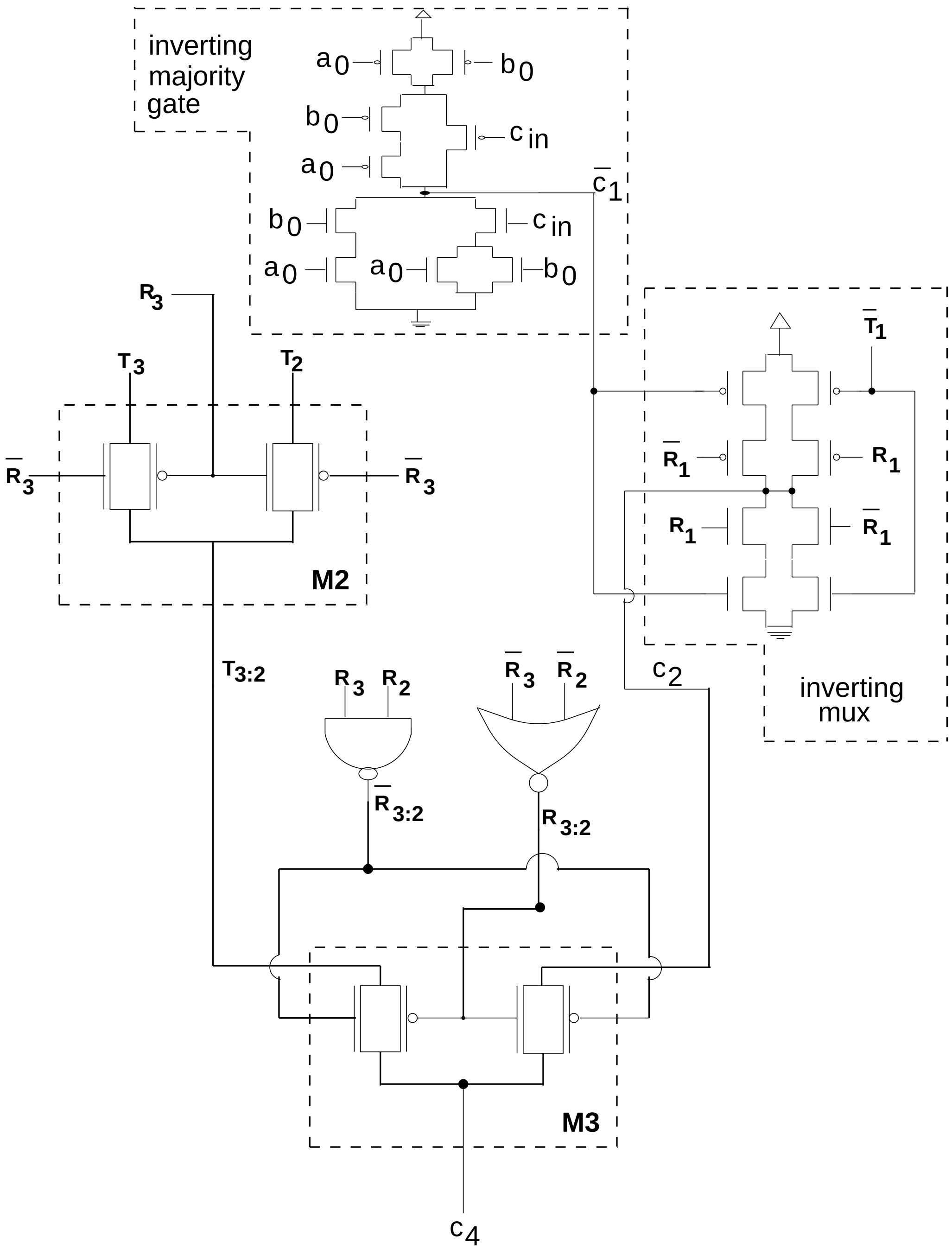
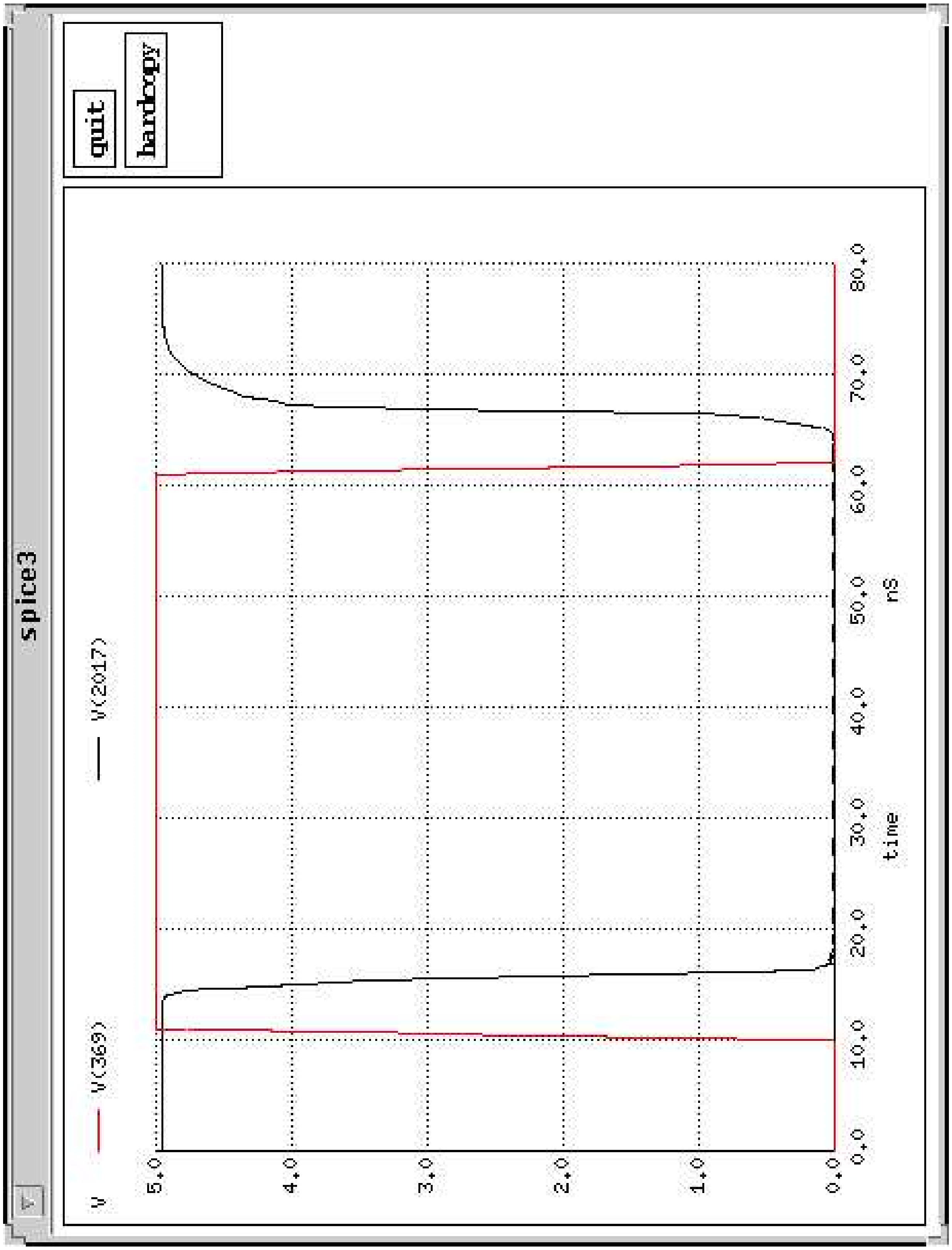


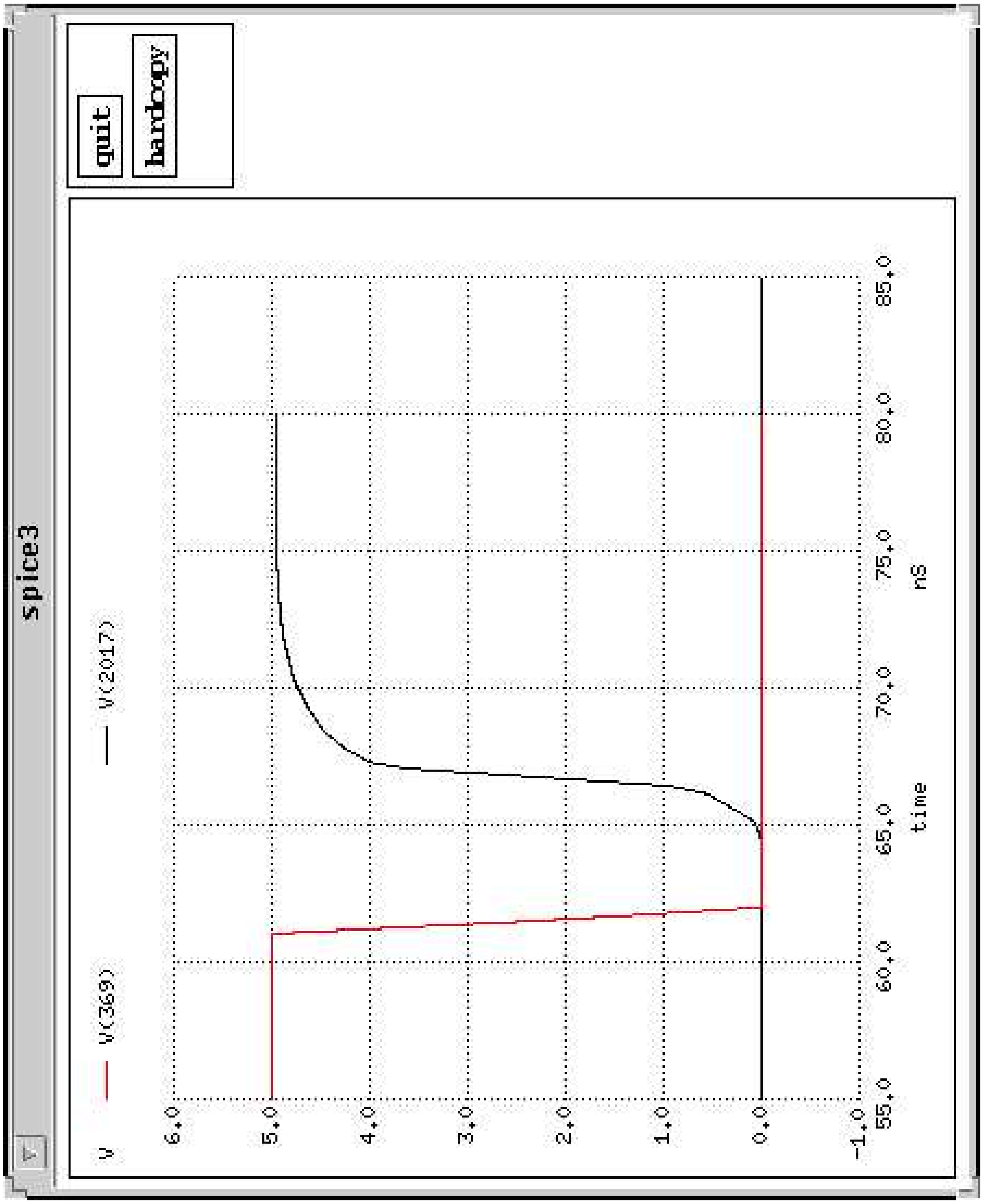
Figure 9 : The "Carry Generate" (CG) block for the least significant bit-group. This is used instead of the SSC block to reduce the critical path delay.

Additional Figures (Spice timing plots)

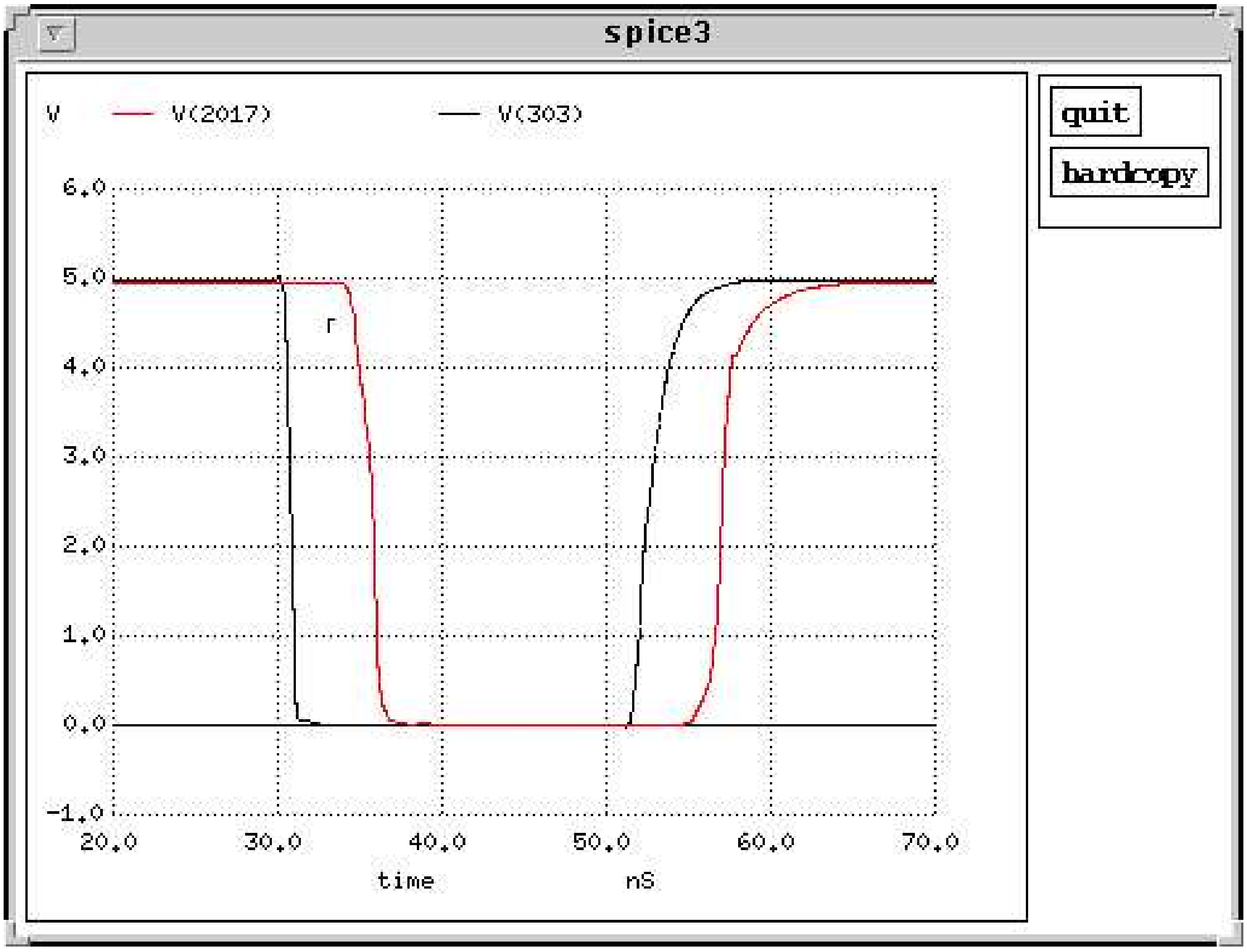
Generated by Hansoo Kim who left and is now at Cadence.
A variant of 32 bit the architure illustrated in the paper was layed out in MAGIC, extracted with 2 micron technology files and simplated in SPICE (v3f4). The plots are one each on a page with bare minimal explanations at the bottom. This file is concatenated to main.ps by the dvipscat utility in a hurry !



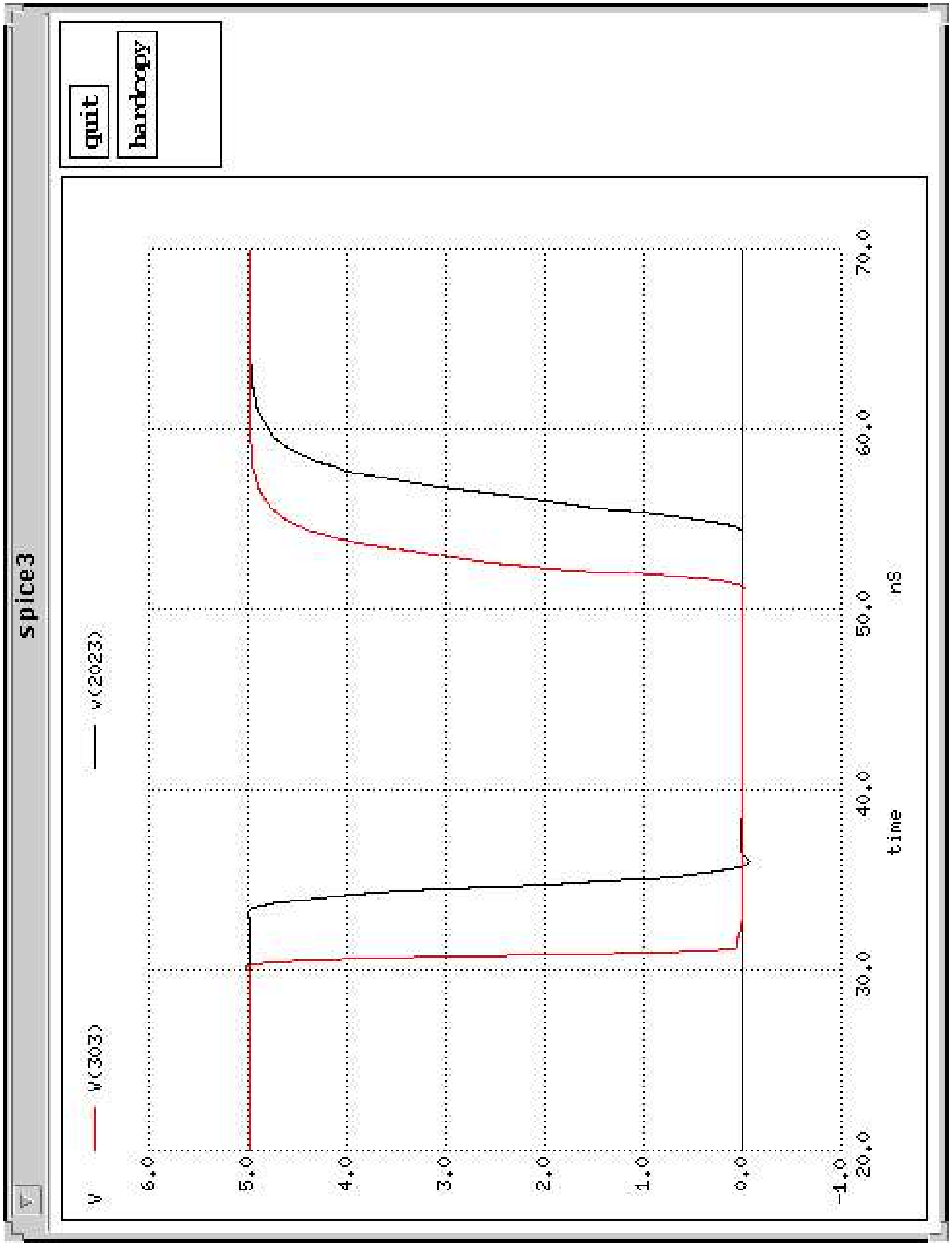
C_{in} to S_{32} delay



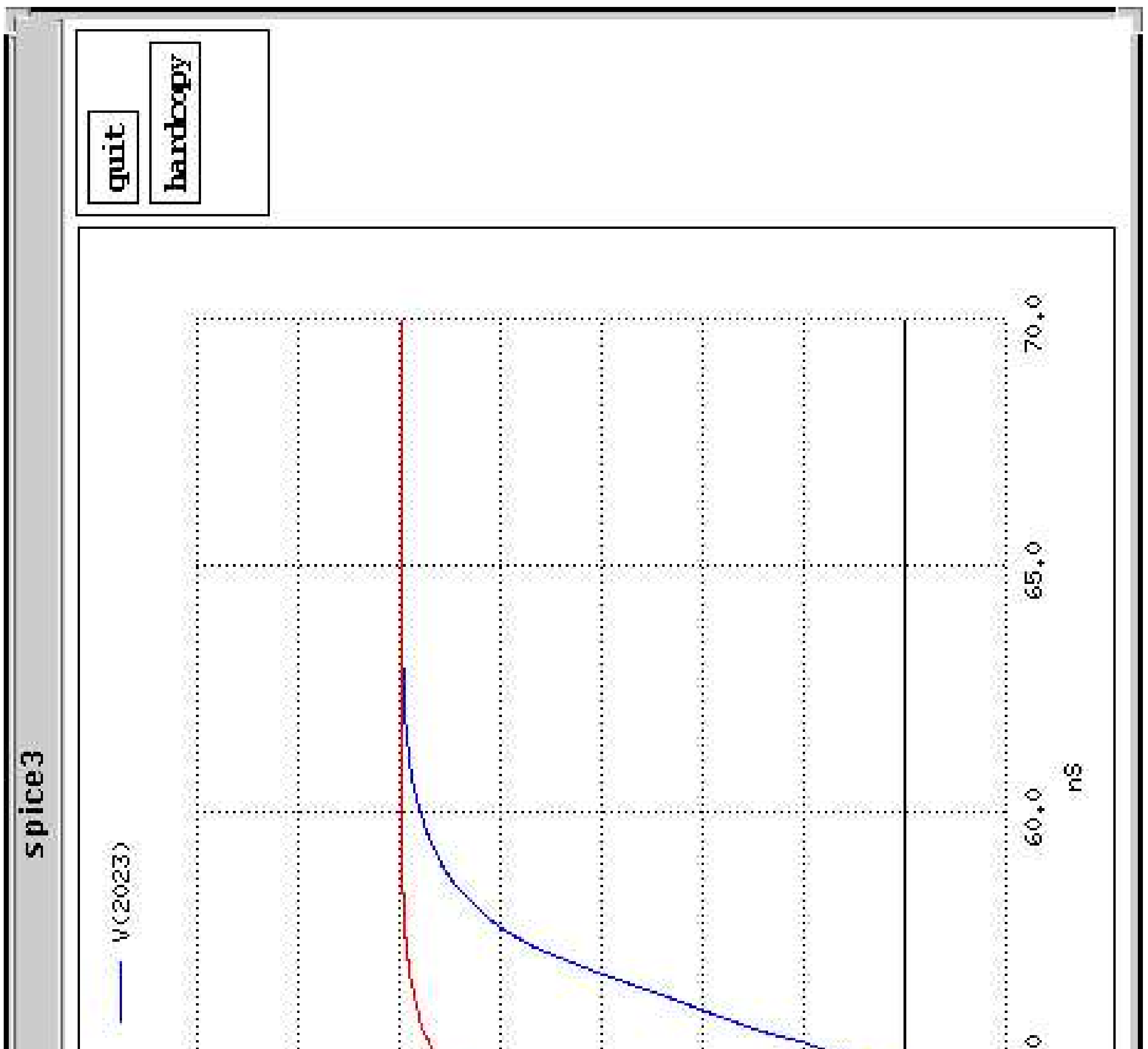
C_{in} to S_{32} pullup (0 to 1 transition) delay



$\bar{P}_4$ to S_{32} delay



$\bar{P}_4$ to $\bar{C}_{28}$ delay


spice3
quit
hardcopy
V
V(303)
V(2023)
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
-1.0
45.0
50.0
55.0
60.0
65.0
70.0
time
nS

$\bar{P}_4$ to $\bar{C}_{28}$ delay detail