Self-Timed Carry-Lookahead Adders

Fu-Chiung Cheng, Stephen H. Unger, *Fellow*, *IEEE*, and Michael Theobald, *Student Member*, *IEEE*

Abstract—Integer addition is one of the most important operations in digital computer systems because the performance of processors is significantly influenced by the speed of their adders. This paper proposes a self-timed carry-lookahead adder in which the logic complexity is a linear function of n, the number of inputs, and the average computation time is proportional to the logarithm of the logarithm of n. To the best of our knowledge, our adder has the best area-time efficiency which is $\Theta(n \log\log n)$. An economic implementation of this adder in CMOS technology is also presented. SPICE simulation results show that, based on random inputs, our 32-bit self-timed carry-lookahead adder is 2.39 and 1.42 times faster than its synchronous counterpart and self-timed ripple-carry adder, respectively; and, based on statistical data gathered from a 32-bit ARM simulator, it is 1.99 and 1.83 times faster than its synchronous counterpart and self-timed ripple-carry adder, respectively.

Index Terms—Self-timed circuits, delay-insenstive circuits, carry-lookahead adders, tree iterative circuits, CMOS.

1 Introduction

Integer addition is one of the most important operations in digital computer systems. In addition to explicit arithmetic (such as addition, subtraction, multiplication, and division) performed in a program, additions are performed to increment program counters and calculate effective addresses [1]. Statistics presented in [1], [2] show that, in a prototypical RISC machine (DLX), 72 percent of the instructions perform additions (or subtractions) in the datapath. The statistics reported in ARM processors even reaches 80 percent [3]. Thus, the performance of processors is significantly influenced by the speed of their adders.

Circuits may be classified as synchronous or asynchronous. Synchronous circuits have a clock to synchronize the operations of subsystems, while asynchronous circuits do not. Subsystems in asynchronous circuits usually need *start* and *completion* mechanisms to synchronize with one another. One advantage of using asynchronous circuits is that these circuits operate at average rates, while synchronous circuits are required to operate at the worst rates. A good example for this is that n-bit ripple-carry adders (which are synchronous), shown in Fig. 1, have worst case computation time $\Theta(n)$,[1] whereas n-bit carry-completion sensing adders [4], [5], [6] (which are asynchronous), shown in Fig. 3, have average computation time $\Theta(\log n)$ [7].

This paper proposes a self-timed carry-lookahead adder in which the logic complexity is a linear function of n, $\Theta(n)$, and the average computation time is proportional to the logarithm of the logarithm of n, $\Theta(\log\log n)$. To the best of our knowledge, our adder has the best area-time efficiency, which is $\Theta(n \log\log n)$.

This paper is organized as follows: Section 2 introduces designs of some earlier synchronous and asynchronous adders. Section 3 presents a novel delay-insensitive carry-lookahead adder with speed-up circuitry. The complexity analysis is presented in Section 4. Section 5 shows the economical CMOS implementation of the propsosed adder. Section 6 presents the results of the SPICE simulation based on both random and statistical data. Section 7 concludes this paper as a whole.

1. In this paper, $\Theta(n)$ ($O(n)/\Omega(n)$) denotes that the complexity or computation time is exactly (at most/at least) a linear function of n.

- *F.-C. Cheng is with the Department of Computer Science and Engineering, Tatung University, Taipei, Taiwan 104, ROC. E-mail: cheng@cse.ttu.edu.tw.*
- *S.H. Unger and M. Theobald are with the Department of Computer Science, Columbia University, New York, NY 10027. E-mail: {unger, theobald}@cs.columbia.edu.*

Manuscript received 1 Sept. 1999; revised 1 Feb. 2000; accepted 10 Mar. 2000. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111787.

2 Background

Adders may be implemented with synchronous or asynchronous circuits. This section introduces some previous work on adder design, including both synchronous and asynchronous adders.

Let $A_{n-1}A_{n-2}\ldots A_0$ and $B_{n-1}B_{n-2}\ldots B_0$ be two n-bit binary numbers with a sum of $S_{n-1}S_{n-2}\ldots S_0$ and with a sequence of carry bits, $C_nC_{n-1}\ldots C_0$. (The least significant bits are A_0 and B_0.)

2.1 Synchronous Adders

2.1.1 Ripple-Carry Adders

The ripple-carry scheme computes the S_is as follows:

$$\begin{aligned} C_{i+1} &= A_iB_i + (A_i + B_i)C_i \\ S_i &= A_i \oplus B_i \oplus C_i, \end{aligned} \tag{1}$$

where $i = 0, 1, \ldots, n-1$ and $\oplus$ is an exclusive OR operation. An n-bit ripple-carry adder (RCA) is constructed by cascading n one-stage full adders, shown in Fig. 1. Obviously, both the logic complexity and the worst case computation time are $\Theta(n)$.

2.1.2 Carry-Lookahead Adders

To compute a sum, an RCA requires, in the worst case, n stage-propagation delays. For high speed processors, this scheme is undesirable. One way to improve adder performance is to use parallel processing in computing

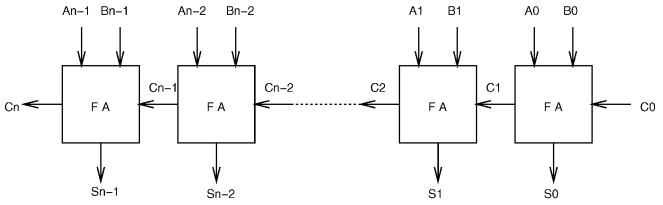


Fig. 1. Ripple-carry adder.

the carries. That is why Carry-Lookahead Adders are introduced. The carry-lookahead scheme computes the $S_i s$ as follows:

$$p_i = A_i \oplus B_i \quad (carry - propagate) \tag{2}$$

$$g_i = A_i B_i \qquad (carry - generate) \tag{3}$$

$$S_i = A_i \oplus B_i \oplus C_i \tag{4}$$

$$C_{i+1} = g_i + p_i C_i, \tag{5}$$

where $i = 0, 1, ..., n-1$. Equation (5) may be further expanded into

$$C_{i+1} = g_i + p_i g_{i-1} + \ldots + p_i p_{i-1} \ldots p_1 g_0 + p_i p_{i-1} \ldots p_0 C_0. \tag{6}$$

For large i, it is impractical to build a two-stage full carry-lookahead adder because of the practical limitations on fan-in and fan-out, irregular structure, and many long wires [8], [1]. However, the carry-lookahead scheme may be built in the form of a tree-like circuit, which has a simple, regular structure [9], [10], [1], by reformulating (6) into

$$P_{i,k} = P_{i,j} P_{j-1,k} \qquad (block - carry - propagate) \tag{7}$$

$$G_{i,k} = G_{i,j} + P_{i,j} G_{j-1,k} \quad (block - carry - generate) \tag{8}$$

$$C_j = G_{j-1,k} + P_{j-1,k} C_k, \tag{9}$$

where $i \geq j > k$, $G_{i,i} = g_i$, and $P_{i,i} = p_i$.

The tree-like circuit of the CLA with $n = 8$ is shown in Fig. 2c. It consists of two kinds of modules: A and B. A-modules, shown in Fig. 2a, compute *carry-propagate* (see (2)), *carry-generate* (see (3)), and sum bits (see (4)). The inputs A_i and B_i to the ith A-module represent the ith bits of binary numbers, which are to be added. The outputs g_i and p_i are the ith *carry-generate* and *carry-propagate*, respectively.

B-modules, shown in Fig. 2b, compute *block-carry-propagate* (see (7)), *block-carry-generate* (see (8)), and carry bits (see (9)).

In a synchronous CPU with a CLA, the addition operation is synchronized by clock pulses. The clock period must be large enough to allow all the possible input configurations to be computed. Thus, the CLA must work

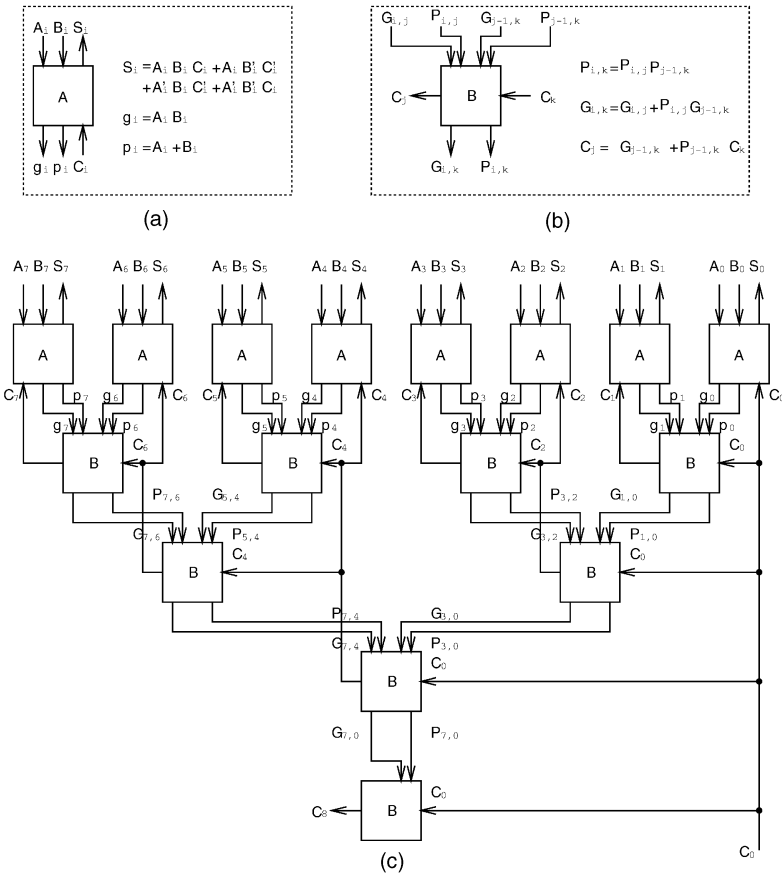


Fig. 2. Tree circuit of carry lookahead adder. (a) A-module. (b) B-module. (c) The circuit of CLA.

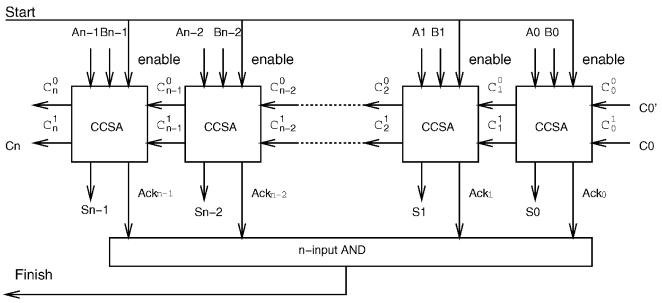


Fig. 3. Carry-completion sensing adder.

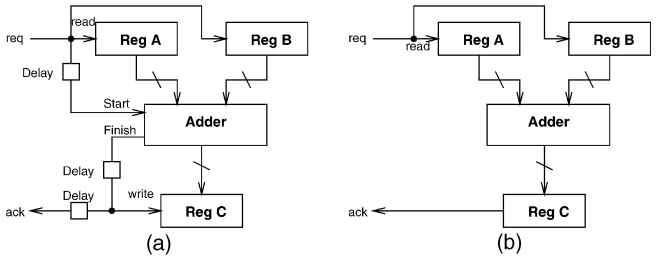


Fig. 4. Data communication of asynchronous adders. (a) Asynchronous adder with bundling constraint. (b) Delay-insensitive adder.

under the worst possible condition. The worst propagation delay of an n-bit CLA is two unit delays[2] of the A-module plus $2\log_2 n - 1$ unit delays of the B-module.

2.1.3 Other Tree-Like Adders

Conditional-Sum Adders [11], Type-2 Adders [10], and Brent and Kung Adders (BKA) [12] were proposed to further improve the worst case delay. The worst propagation delay of these adders is about $(\log_2 n)$-stage delays. However, the logic complexity of these adders goes up to $\Theta(n \log n)$.

2.2 Asynchronous Adders

2.2.1 Carry-Completion Sensing Adders

A Carry-Completion Sensing Adder (CCSA) [4], [5], [13] may be regarded as an asynchronous version of an RCA. Instead of using clock pulses to synchronize adder operation, a CCSA uses some extra circuitry to implement the start and completion signals. Fig. 3 shows an n-bit CCSA. The CCSA scheme computes the $S_i s$ in the following way:

$$\begin{aligned} C^0_{i+1} &= enable(A'_i B'_i + (A'_i + B'_i)C^0_i) \\ C^1_{i+1} &= enable(A_i B_i + (A_i + B_i)C^1_i) \\ Ack_i &= C^0_{i+1} + C^1_{i+1} \\ S_i &= A_i \oplus B_i \oplus C_i, \end{aligned}$$

where $i = 0, 1, \ldots, n-1$, x' is $\bar{x}$, $C^0_0 = C'_0$, and $C^1_0 = C_0$.

The primary inputs, A_i and B_i, are single rail. The carry bits are encoded by using two separate signals (dual-rail signaling): $C^0_{i+1} = 1$ means that no carry emerges from the ith one-bit adder and $C^1_{i+1} = 1$ means that a carry emerges from the ith one-bit adder. The completion or acknowledge signal (Ack_i) for each stage is turned on by the arrival of either C^0_{i+1} or C^1_{i+1}. This is most economically detected by an OR-gate. If desired, an AND-gate can be added to flag the error condition $C^0_{i+1} = C^1_{i+1} = 1$.

Once all the stages have computed their carries, the addition is completed. An n-input AND gate may be used to signal the completion (i.e., $finish = \prod_{i=0}^{n-1} Ack_i$).

The *enable* signal is used to start the computation and to ensure that no false completion signal will be generated. When $enable = 0$, all C^0_i and C^1_i $(i = 1, ..n)$ signals are set to zero. The completion signal, *finish*, must be zero, too. Thus, no false completion can be asserted. After all the input data have arrived at the input ports of the CCSA, the *enable* signal is turned on to start the addition operation. Upon the completion of the addition, the *finish* signal is turned on.

For more detail on the CCSA, see [4], [5], [13]. The logic complexity is $\Theta(n)$ and the average computation time for randomly distributed inputs is $\Theta(\log n)$ [7]. Note that the worst case computation time is $\Theta(n)$.

2.2.2 Delay-Insensitive Circuits

Delay-insensitive (DI) circuits [14] are a subclass of asynchronous circuits. The defining property of DI circuits is that their correctness is insensitive to delays in both gate elements and connection wires. Thus, DI circuits are the most robust circuits in terms of the operating variations such as temperature, voltage, and processing. The class of pure DI circuits is quite limited [15]. However, extending pure DI circuits with isochronic forks is sufficient to construct any circuit of interest. (Such circuits are sometimes called quasi-DI.) For this paper, we assume isochronic forks.

The CCSA, shown in Fig. 3, is not a DI circuit. It must meet the bundling constraint [16], [13]: The *start* signal cannot be asserted unless all the input data bits have arrived and the sum bits must arrive at the environment before the environment receives the *finish* signal.

One way to meet the above constraint is to put a delay in the control path. Fig. 4a shows an example of how an adder with bundling constraint is constructed. It works as follows: First, the *req* signal loads the data of the registers A and B to the input ports of the adder and then the *start* signal is turned on. The delay in the path from *req* to *start* guarantees that all the data bits of the registers A and B have arrived at the input ports of the adder before the adder starts its computation. Second, the sum bits have arrived at the register C and then the register C is to store them. The delay in the path from *finish* to *write* guarantees that all the sum bits are propagated faster to the register C than the *finish* signal is. Finally, an *ack* signal is generated to indicate the completion of an addition operation. The *req*, *start*, *finish*, *write*, and *ack* signals must be reset to zero before the next addition starts.

It is impossible to meet bundling constraints if there is no way to control delays in control and data paths. Besides, DI circuits are the most robust circuits and particularly easy to compose and substitute. Thus, the design of DI circuits is very interesting.

2. One unit delay of A- or B-modules is equal to the delay of two *AND* or *OR* gates if they are implemented in two-level logic.

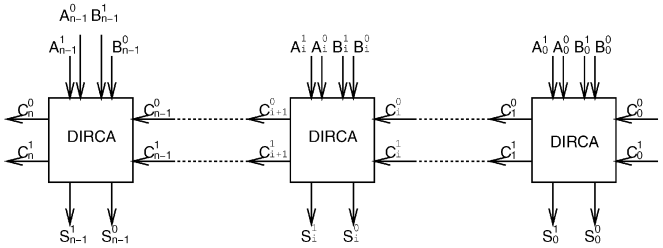


Fig. 5. Delay-insensitive ripple-carry adder.

2.2.3 Delay-Insensitive Ripple-Carry Adders

To implement DI ripple-carry adders (DIRCA), we use dual-rail signaling for input, sum, and carry bits. DIRCA computes the $S_i s$ in the following way:

$$C_{i+1}^0 = A_i^0 B_i^0 + (A_i^0 + B_i^0) C_i^0 \tag{10}$$

$$C_{i+1}^1 = A_i^1 B_i^1 + (A_i^1 + B_i^1) C_i^1 \tag{11}$$

$$S_i^0 = A_i^0 B_i^0 C_i^0 + A_i^0 B_i^1 C_i^1 + A_i^1 B_i^0 C_i^1 + A_i^1 B_i^1 C_i^0 \tag{12}$$

$$S_i^1 = A_i^1 B_i^1 C_i^1 + A_i^1 B_i^0 C_i^0 + A_i^0 B_i^1 C_i^0 + A_i^0 B_i^0 C_i^1, \tag{13}$$

where $i = 0, 1, \ldots, n-1$.

An n-bit DIRCA is shown in Fig. 5. The logic complexity of DIRCA is a linear function of n, $\Theta(n)$, and the average computation time is proportional to the logarithm of n, $\Theta(\log n)$.

Fig. 4b shows an example of how a DI adder is constructed. It works as follows: The *req* signal loads the data of registers A and B, taking them to the input ports of the adder. The DI adder computes sum bits whenever any data bits arrive. Any sum bits produced by the DI adder are sent to and stored in register C. Once the register C receives all the sum bits, it sends out the *ack* signal to indicate the completion of an addition operation. The *req*, *ack*, and *the input and output signals of the adder* must be reset to zero before next addition starts. The registers used by the DI adder are dual-rail asynchronous registers in [17].

Martin [18] proposed a very good design of DIRCA adder by using CMOS technology. The transistor count per DIRCA cell is 42. Compared to the synchronous RCA cell which needs 40 transistors, it is clear that the asynchronous DIRCA adder is hardly larger than the synchronous one, in spite of the use of dual-rail signals.

In the next section, we will present a delay-insensitive carry-lookahead adder in which the logic complexity is $\Theta(n)$ and the time complexity is $\Theta(\log \log n)$.

3 Delay-Insensitive Carry-Lookahead Adders

Delay-Insensitive Carry-Lookahead Adders (DICLA) may be implemented by using dual-rail signaling in input bits, sum bits, and carry bits, and by using one-hot code in the internal signals. A DICLA may be built with two basic modules: C and D, connected in a tree-like structure (Fig. 6). The equations of the C-module are defined as follows:

$$k_i = A_i^0 B_i^0 \qquad (carry - kill) \tag{14}$$

$$g_i = A_i^1 B_i^1 \qquad (carry - generate) \tag{15}$$

$$p_i = A_i^0 B_i^1 + A_i^1 B_i^0 \qquad (carry - propagate) \tag{16}$$

$$S_i^0 = A_i^0 B_i^0 C_i^0 + A_i^1 B_i^1 C_i^0 + A_i^0 B_i^1 C_i^1 + A_i^1 B_i^0 C_i^1 \tag{17}$$

$$S_i^1 = A_i^1 B_i^1 C_i^1 + A_i^1 B_i^0 C_i^0 + A_i^0 B_i^1 C_i^0 + A_i^0 B_i^0 C_i^1, \tag{18}$$

where $i = 0, .., n-1$.

Note that, in a DICLA, the input data bits and the output data (sum) bits may be propagated at any time in any order. The completion signal, *finish*, may be generated when needed as follows:

$$finish = (C_n^0 + C_n^1) \prod_{i=0}^{n-1} (S_i^0 + S_i^1).$$

For more completion detection circuits for dual-rail self-timed systems, see [19].

The C-module is shown in Fig. 6a. The dual-rail signals on the lefthand side of Fig. 6a are grouped as $A_i = (A_i^0, A_i^1)$, $B_i = (B_i^0, B_i^1)$, $C_i = (C_i^0, C_i^1)$, $S_i = (S_i^0, S_i^1)$, and $I_i = (k_i, g_i, p_i)$. The resulting C-module is shown on the righthand side of Fig. 6a.

The equations for the D-module are defined as follows:

$$P_{i,k} = P_{i,j} P_{j-1,k} \qquad (block - carry - propagate) \tag{19}$$

$$K_{i,k} = K_{i,j} + P_{i,j} K_{j-1,k} \qquad (block - carry - kill) \tag{20}$$

$$G_{i,k} = G_{i,j} + P_{i,j} G_{j-1,k} \qquad (block - carry - generate) \tag{21}$$

$$C_j^0 = K_{j-1,k} + P_{j-1,k} C_k^0 \tag{22}$$

$$C_j^1 = G_{j-1,k} + P_{j-1,k} C_k^1. \tag{23}$$

The D-module is shown in Fig. 6b. The signals on the lefthand side of Fig. 6b are grouped as $I_{i,j} = (K_{i,j}, G_{i,j}, P_{i,j})$ and $C_i = (C_i^0, C_i^1)$. The resulting D-module is shown on the righthand side of Fig. 6b.

Initially, all carries (i.e., C_i^0 and C_i^1 for $i = 1, 2, \ldots, n$) and the internal signals (i.e., $K_{i,j}$, $G_{i,j}$, and $P_{i,j}$) are zero because all primary inputs (i.e., A_i^0, A_i^1, B_i^0, and B_i^1 for $i = 0, 1, \ldots, n-1$) and input carry (i.e., C_0^0 and C_0^1) are zero. During the computation, C_i^0 and C_i^1 will not be both turned on simultaneously and exactly one of the internal signals will be turned on.

The performance of the DICLA may be further improved by some speed-up circuitry. It is obvious that if $A_i = B_i$ (i.e., carry-kill or carry-generate), then the output carry, C_{i+1}, is independent of the input carry, C_i. The tree-like circuit, shown in Fig. 6, does not take full advantage of this feature to speed up the carry computation.

3.1 Speed-Up Circuitry

The idea of speeding up carry computation of tree-like adders is to send the *carry-generate*s and *carry-kill*s to all

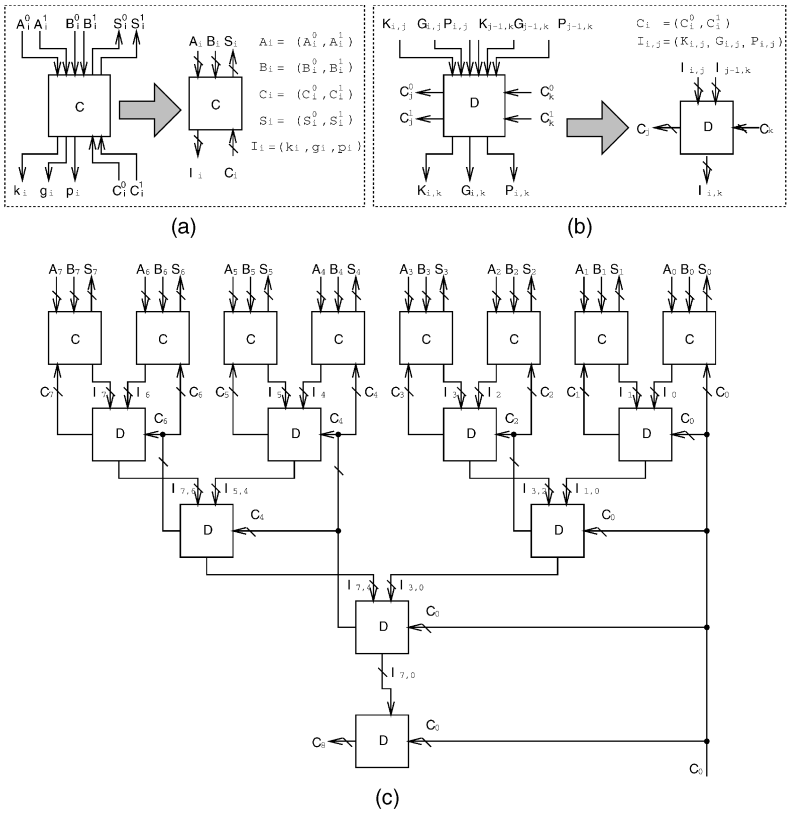


Fig. 6. Delay-insensitive carry-lookahead adder.

stages that can use this information directly. For example, the DICLA, shown in Fig. 6, takes one unit delay of the C-module plus three unit delays of the D-module to compute C_4. However, C_4^1 may sometimes be determined by g_3, $G_{3,2}$, or $G_{3,0}$. If we know $A_3^1 = B_3^1 = 1$ (i.e., $g_3 = 1$), we may directly set $C_4^1 = g_3$. Thus, it takes one unit delay of the C-module plus only one unit delay of the D-module to compute C_4^1.

The D'-module (D-module with speed-up circuitry), shown in Fig. 7a, is designed to speed up the carry computation. It is defined in the following way:

$$P_{i,k} = P_{i,j}P_{j-1,k} \tag{24}$$

$$K_{i,k} = K_{i,j} + P_{i,j}K_{j-1,k} \tag{25}$$

$$G_{i,k} = G_{i,j} + P_{i,j}G_{j-1,k} \tag{26}$$

$$C_j^0 = Z_{j-1}^0 + K_{j-1,k} + P_{j-1,k}C_k^0 \tag{27}$$

$$C_j^1 = Z_{j-1}^1 + G_{j-1,k} + P_{j-1,k}C_k^1. \tag{28}$$

The definitions of the *block-carry-propagation*, *block-carry-kill*, and *block-carry-generate* of the D'-module are the same as those of the D-module. However, each output carry (C_j^0/C_j^1) has an extra speed-up signal (Z_{j-1}^0/Z_{j-1}^1).

The general form of speed-up circuits is defined as follows:

$$Z_{j-1}^0 = k_{j-1} + \sum_{x=0}^{j-2} k_x \prod_{y=x+1}^{j-1} p_y \tag{29}$$

$$Z_{j-1}^1 = g_{j-1} + \sum_{x=0}^{j-2} g_x \prod_{y=x+1}^{j-1} p_y, \tag{30}$$

where $2 \le j \le n$. These two equations are taken from the scheme of full carry-lookahead adders.

It is impractical to implement the speed-up circuits, shown in (29) and (30), when n is large because of the practical limitations on fan-in and fan-out, irregular structure, and many long wires. In addition, the logic complexity of the speed-up circuits increases more than linearly (i.e., $\Theta(n \log n)$).

Fortunately, all the above problems can be solved by using the properties of a tree-like structure. That is, instead of fully employing the carry-lookahead generation to each carry, our speed-up mechanism focuses on the root nodes of a subtree. By trading time with area, the logic complexity of the adder with the speed-up circuitry is brought down to a linear function of n, the number of inputs, and the average computation time is proportional to $\Theta(\log \log n)$. Both proofs will be given in Section 4.

The above two equations can be reformulated in a more efficient and economical way in logic as:

$$Z_{j-1}^0 = k_{j-1} + \sum_{x=1}^{l-2} K_{j-1,j-2^x} \tag{31}$$

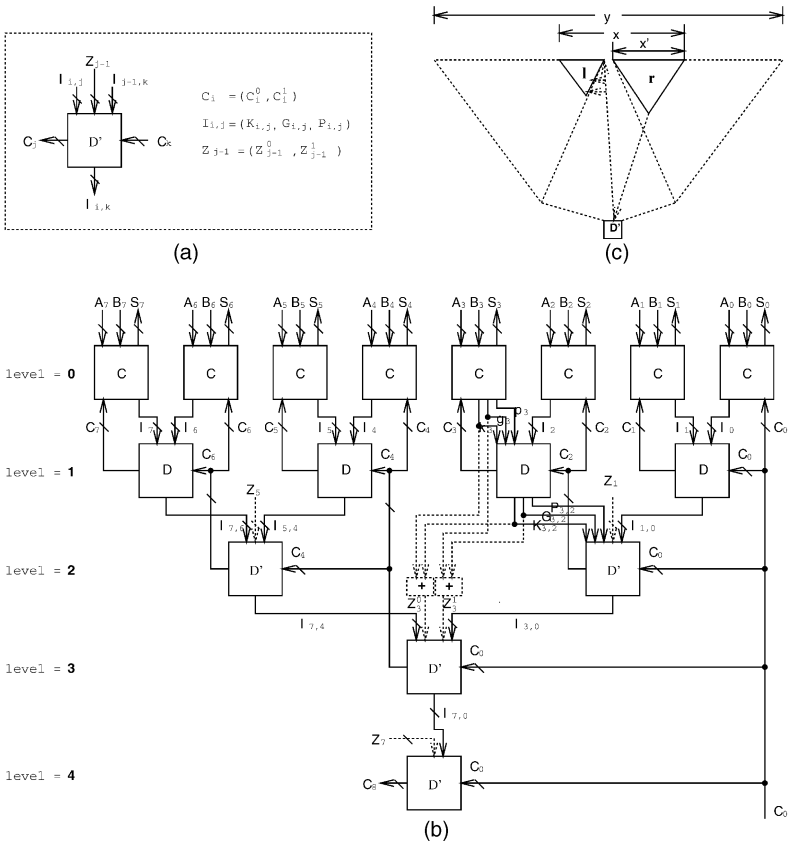


Fig. 7. D'-module and delay-insensitive carry-lookahead adder with speed-up circuits. (a) D'-module. (b) DICLA with speed-up circuitry (SPDICLA). (c) A carry chain with length = x.

$$Z^1_{j-1} = g_{j-1} + \sum_{x=1}^{l-2} G_{j-1,j-2^x}, \tag{32}$$

where $j = 2, 4, 6, \ldots, n$ and l is the level of the D'-module, receiving the speed-up signals, in the tree. The tree leaves are at level 0 and the immediate parent nodes of nodes at level i are at level $i+1$ (Fig. 7b). Note that only the modules at $level \geq 2$ need speed-up circuits.

In this speed-up mechanism, each root node above level 1 receives speed-up signals from the left edge nodes of its right subtree. For example, the D'-module at level 3 in Fig. 7b has the following speed-up signals: $Z^1_3 = g_3 + G_{3,2}$ and $Z^0_3 = k_3 + K_{3,2}$. The DICLA with the speed-up circuitry (SPDICLA) for $n = 8$ is shown in Fig. 7b. The speedup circuitry is shown in dotted lines, and Z^1_3 and Z^0_3 are shown in detail.

Fig. 7c is used to demonstrate the power of the speed-up circuitry. Imagine a carry propagation chain (i.e., a sequence of carry-propagates) with length $= x$ which is spanned by two subtrees, shown in the solid triangles of Fig. 7c. The **r** (right) subtree has a carry propagation chain with length $= x'$ and the **l** (left) subtree has a carry propagation chain with length = $x - x'$. Assume that x' and $x - x'$ are powers of 2 and $x' \geq x - x'$. There must exist two subtrees with the same size (say $\frac{y}{2}$) which contain the **r** and **l** subtrees. It takes $2\log_2 x' + 1$ module delays to compute the carries in the **r** subtree. Thanks to the speed-up circuitry, the carries can be directly propagated from the right subtree to the left subtree. The **l** subtree can start carry computation at $\log_2 x' + 1$ (the computation time from the leaves to the root node of r subtree) plus 1 (the delay of the D'-module which is the root of the two dotted subtrees) delays. Thus, it takes only

$$\log_2 x' + 1 + Max(\log_2 x', 1 + \log_2(x - x') + 1)$$

delays to compute all these carries. Without the speed-up circuitry, it would take $\log_2 y + 1 + \log_2(x - x') + 1$ delays.

For any n-bit addition with a maximal carry chain length = c, which is a small constant, if the carry chain is located in the middle of the tree, then it requires $\Theta(\log n)$ stage delays in DICLA, but only $\Theta(1)$ stage delays in SPDICLA.

3.2 Optimization

Adding speed-up circuitry not only enhances the performance, but also makes some logic redundant and, hence, removable. Namely, if the carry-kills and carry-generates are exploited in the speed-up circuitry, these signals need not propagate through the tree. For example, since g_3 and k_3 are used by the D'-module, which computes C_4, it is unnecessary to route them to the D-module, which computes C_3. In general, the left carry-kill and left carry-generate signals of D- and D'-modules can be eliminated.

The equations for the simplified D-module (SD) are redefined as follows:

$$P_{i,k} = P_{i,j} P_{j-1,k} \tag{33}$$

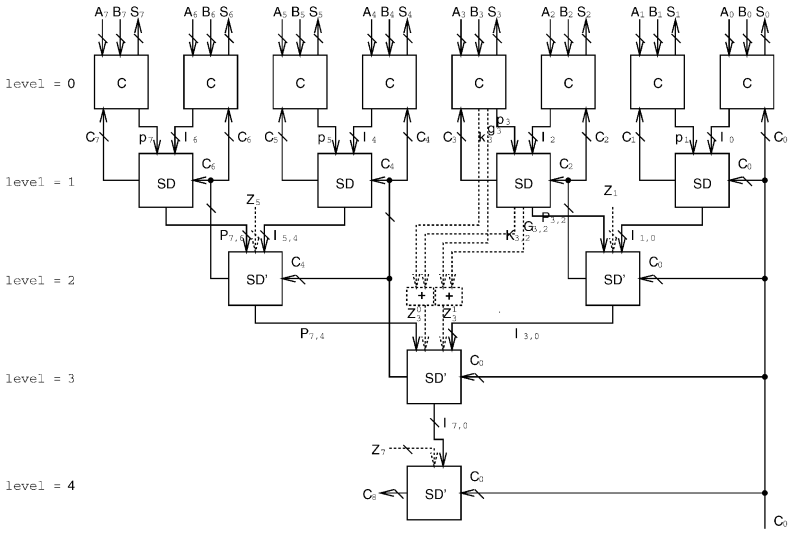


Fig. 8. DICLASP.

$$K_{i,k} = P_{i,j}K_{j-1,k} \tag{34}$$

$$G_{i,k} = P_{i,j}G_{j-1,k} \tag{35}$$

$$C_j^0 = K_{j-1,k} + P_{j-1,k}C_k^0 \tag{36}$$

$$C_j^1 = G_{j-1,k} + P_{j-1,k}C_k^1. \tag{37}$$

Note that $P_{i,k}$, C_j^0, and C_j^1 are not changed. And, the equations for the simplified D'-module (SD') are redefined as follows:

$$P_{i,k} = P_{i,j}P_{j-1,k} \tag{38}$$

$$K_{i,k} = P_{i,j}(Z_{j-1}^0 + K_{j-1,k}) \tag{39}$$

$$G_{i,k} = P_{i,j}(Z_{j-1}^1 + G_{j-1,k}) \tag{40}$$

$$C_j^0 = Z_{j-1}^0 + K_{j-1,k} + P_{j-1,k}C_k^0 \tag{41}$$

$$C_j^1 = Z_{j-1}^1 + G_{j-1,k} + P_{j-1,k}C_k^1. \tag{42}$$

Note that $P_{i,k}$, C_j^0, and C_j^1 are not changed.

An 8-bit DICLA with SD- and SD'-modules and the speed-up circuitry (DICLASP) is shown in Fig. 8.

Note that the SPDICLA is not delay-insensitive due to the speed-up circuitry, which creates multiple paths to compute the K and G signals of D'-modules. This is not a problem in the computation phase. However, in the reset phase, assume that there is a path with long delay and the signal has not asserted by this path. A reset completion may be falsely flagged. Removing the redundant logic, in fact, makes the circuit (i.e., DICLASP) delay-insensitive again.

4 Performance Analysis

In this section, we analyze the logic complexity and the average delay of DICLASP adders. We assume three things: First, the delay through each module (e.g., C-, SD-, and SD'-modules) is d; second, the number of bit positions of input arguments, n, is a power of 2, i.e., $n = 2^k$; third, the distribution of the input configurations is uniform.

4.1 Average Computation Time

The time required to perform an addition (*computation time*) in an adder is the time required for propagating the carries (*carry propagation time*) in stages plus one more delay to compute the sum bits. The computation time of an adder is sensitive to the numbers to be added. The upper and lower bound proofs of average computation time are an extension of proofs for CCSAs by Greenstreet [20].

Theorem 1. *For any input configuration, the carry propagation time is proportional to the logarithm of the length of the longest carry chain.*

Proof. Consider a carry chain with length x in an input configuration where $2^{l-1} < x \leq 2^l$.

Upper bound: If the carry chain is contained in one subtree with the length of leaves $= 2^l$ in DICLASP (see Fig. 9a), then the carries in the carry chain can be computed in $(2l+1)d$. To propagate the last carry of the carry chain to the next stage, it may need two more delays. So, the carry propagation time is at most $(2l+3)d$. Otherwise, the carry chain must span more than one subtree with the length of leaves $= 2^l$ (see Fig. 9b). There must exist two subtrees with the length of leaves $= 2^l$ in DICLASP containing the carry chain. In this case, the propagation time can be computed in at most $(2l+3)d$. Since $2^{l-1} < x$, $l < log_2 x + 1$. Thus, $2l + 3 < 2(log_2 x + 1) + 3 = 2log_2 x + 5$. To summarize, in both cases, the carry propagation time is at most $(2\log_2 x + 5)d$.

Lower bound: There must exist one subtree with the length of leaves $= 2^{l-2}$ in DICLASP such that the leaves of the subtree are contained in the carry chain. Thus, the carry propagation time is at least $(2(l-2)+1)d = (2l-3)d$. Since $x \leq 2^l$, $log_2 x \leq l$. Thus, $2l - 3 \geq 2log_2 x - 3$, i.e., the carry propagation time is at least $(2\log_2 x - 3)d$. □

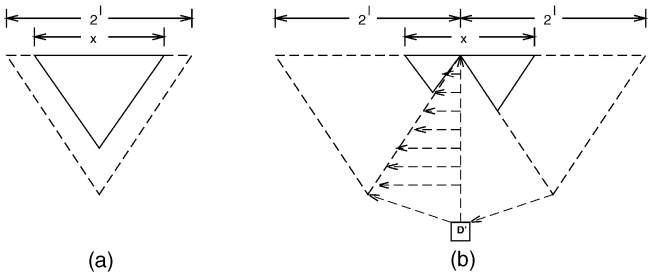


Fig. 9. Spanning of the longest carry chain. (a) Carry chain in one subtree. (b) Carry chain in two subtrees.

Now, we shall show the average computation time of DICLASP is $\Theta(\log\log n)$.

Theorem 2 (Lower Bound). *A lower bound of the average carry propagation time of the DICLASP is* $\Omega(\log\log n)$.

Proof. Partition the n stages into nonoverlapping segments, each of length $\frac{\log_2 n}{2}$. There are $\frac{2n}{\log_2 n}$ such segments.

The probability that a carry propagates across one of these segments is the probability that all $\frac{log_2 n}{2}$ stages in the segment propagate the carry. This is $0.5^{\frac{\log_2 n}{2}} = \frac{1}{\sqrt{n}}$. The probability that a carry is not propagated across a segment is $(1-\frac{1}{\sqrt{n}})$. The probability that there is no segment over which a carry is propagated is $(1-\frac{1}{\sqrt{n}})^{\frac{2n}{\log_2 n}}$.

For $x > 1$, $(1-\frac{1}{x})^x < \frac{1}{e}$, and for $y \geq x > 1$, $(1-\frac{1}{x})^y < \frac{1}{e}$. This is satisfied when $2n \geq \sqrt{n}\log_2 n$, i.e., when $n \geq 1$. So, for $n \geq 1$, the probability that a carry is not propagated across at least one segment is less than $\frac{1}{e}$. This means that the probability that a carry is propagated across at least one segment exceeds $(1-\frac{1}{e})$.

Thus, the average carry propagation time exceeds

$$\left(1-\frac{1}{e}\right)\cdot\left(2\log_2\left(\frac{\log_2 n}{2}\right)-3\right)d = \left(1-\frac{1}{e}\right)\cdot(2\log_2\log_2 n - 5)d,$$

i.e., the average carry propagation time (APT) is $\Omega(\log\log n)$. □

An upper bound is obtained by considering n overlapping length $2\log_2(n)$ chains. We show that such chains are rare.

Theorem 3 (Upper Bound). *An upper bound of the average carry propagation time of the DICLASP is* $O(\log\log n)$.

Proof. Let $P(i)$ be the probability that the longest carry chain has length i and let $T(i)$ be the maximal carry propagation time of chains of length i. Then,

$$\begin{aligned} ACT &\leq \sum_{i=0}^{n} P(i)\cdot T(i) \\ &\leq P(0)\cdot 3d + \sum_{i=1}^{2\log_2 n} P(i)\cdot(2\log_2 i + 5)d \\ &\quad + \sum_{i=2\log_2 n+1}^{n} P(i)\cdot(2\log_2 i + 5)d. \end{aligned}$$

The probability that the maximal carry chain = 0 is $\frac{1}{2^n}$. Thus, $P(0)\cdot 3d = \frac{3}{2^n}d$.

The second term can be derived as follows: Since $\sum_{i=1}^{2\log_2 n} P(i) < 1$ and $i \leq 2\log_2 n$

$$\sum_{i=1}^{2\log_2 n} P(i)\cdot(2\log_2 i + 5)d < \sum_{i=1}^{2\log_2 n} P(i)\cdot(2\log_2(2\log_2 n) + 5)d$$
$$< (2\log_2(2\log_2 n) + 5)d = (2\log_2\log_2 n + 7)d.$$

The third term can be derived by considering n overlapping segments of length $2\log_2 n$. Each carry chain of length longer than $2\log_2 n$ contains at least one segment completely. The probability that a carry propagates across such a segment is $(0.5)^{2\log_2 n} = \frac{1}{n^2}$. The number of segments, even allowing overlapping, is less than n. Also, the length of any carry chain is at most n. Thus, the contribution to the expected carry propagation time of carry chains that are longer than $2\log_2 n$ is less than $n\cdot\frac{1}{n^2}\cdot(2\log_2 n + 5)d = \frac{2\log_2 n+5}{n}d$.

Since $APT < (\frac{3}{2^n} + 2\log_2\log_2 n + 7 + \frac{2\log_2 n+5}{n})d$, the average carry propagation time is $O(\log\log n)$. □

Theorem 4 (Average Computation Time). *The average computation time of the DICLASP is* $\Theta(\log\log n)$.

Proof. By Theorems 2 and 3. □

4.2 Logic Complexity

Theorem 5. *The logic complexity of DICLASP is a linear function of* n.

Proof. It is easy to show that the n-input DICLASP requires n C-modules, $\frac{n}{2}$ SD-modules, $\frac{n}{2}$ SD'-modules and the speed-up circuitry. We show that the speed-up circuitry is also a linear function of n by counting the total number of added inputs.

Consider the SD'-modules at level i, where $i \leq k$ ($k = \log_2 n$). There are 2^{k-i} SD'-modules and each SD'-module at level i has $2(i-1)$ inputs (i.e., $i-1$ inputs for carry generate and $i-1$ for carry kill) for the speed-up circuitry. We also need the speed-up circuit, which contains $2k$ inputs, to compute the last carry. The total number of the added inputs is

$$N = 2(k + \sum_{i=2}^{k} 2^{k-i}(i-1)) = 2n - 2.$$

□

For VLSI implementation, the fan-in and fan-out for C- and SD-modules used in DICLASP are no problem at all. The maximal fan-in and fan-out of SD'-modules is $\log_2 n$. This is not a severe problem either, except for a very large n.

TABLE 1
Logic and Time Complexity of Adders

Type of Adder	Circuit	Logic Complexity	Computation Time	Area-Time Efficiency
RCA	Sync	$O(n)$	$O(n)$	$O(n^2)$
CCSA	Async (BC)	$O(n)$	$O(\log n)$	$O(n \log n)$
DIRCA	Async (DI)	$O(n)$	$O(\log n)$	$O(n \log n)$
CSA1	Sync	$O(n \log n)$	$O(\log n)$	$O(n(\log n)^2)$
CSA2	Sync	$O(n)$	$O(\sqrt{n})$	$O(n\sqrt{n})$
CSA3	Sync	$O(n)$	$O(\sqrt{n})$	$O(n\sqrt{n})$
Type-2	Sync	$O(n \log n)$	$O(\log n)$	$O(n(\log n)^2)$
BKA	Sync	$O(n \log n)$	$O(\log n)$	$O(n(\log n)^2)$
CSCSA	Async (ST)	$O(n \log n)$	$O(\log \log n)$	$O(n \log n \log \log n)$
CLA	Sync	$O(n)$	$O(\log n)$	$O(n \log n)$
DICLA	Async (DI)	$O(n)$	$O(\log n)$	$O(n \log n)$
SPDICLA	Async (ST)	$O(n)$	$O(\log \log n)$	$O(n \log \log n)$
DICLASP	Async (DI)	$O(n)$	$O(\log \log n)$	$O(n \log \log n)$

The merged cell method [10], which merges two or more primary modules (e.g., C-modules) or two or more interior modules (e.g., D- and D'-modules) to reduce the depth of the tree, may be applied further to speed up the addition operation.

4.3 Comparisons of Adders

In this section, we compare the logic complexity and computation time for both synchronous and asynchronous adders. They are Ripple Carry Adder (RCA), Conditional-Sum Adder(CSA1) [11], Carry-Select Adder(CSA2) [21], [1], Carry-Skip Adder (CSA3) [22], [1], Carry-Completion Sensing Adder (CCSA) [4], Delay-Insensitive Ripple Carry Adder (DIRCA) [18], Conditional-Sum Completion-Sensing Adder (CSCSA) [23], Brent and Kung Carry-Lookahead Adder (BKA) [12], Type-2 Adder [10], and, our DICLA, SPDICLA, and DICLASP.

The logic (area) complexity and computation time of the above adders are listed in Table 1. The worst case computation time is assumed for the synchronous adders and average computation time is assumed for the asynchronous adders. Area-time efficiency is computed by multiplying logic complexity and time complexity.

The asynchronous adders listed in Table 1 are classified as bundling constraint (BC), delay-insensitive (DI), and self-timed (ST) adders. In bundling constraint (BC) and self-timed (ST) adders, delay assumptions have to be made to guarantee correct operations. In a DI adder, input signals may arrive at any time in any order since its correctness is insensitive to delay.

It is worth mentioning that the average computation time of CSCSA is also $O(\log \log n)$. However, the logic complexity of CSCSA is $O(n \log n)$ and the design requires circuits to detect the true sum at each level and to route the completed sum from any arbitrary level to the output latch. Our DICLASP is a delay-insensitive circuit and has the best area-time efficiency among these adders.

4.4 Delay Analysis by Program Simulation

From the lower bound (Theorem 2) and the upper bound (Theorem 3) theorems, we can easily derive that the constant factor of the average computation time of the DICLASP is between 2 and 1.264. It would be very interesting to know the exact constant factor of the time complexity of the DICLASP and to compare it with other adders. No exact mathematical model for the DICLASP has been found yet. Thus, simulation was used to analyze the average computation delay. For simulations of adders, see [2], [24].

The DIRCA, the DICLA, and the DICLASP adders have been simulated with C++ programs. The results are shown in Table 2. The results of 8-bit adders are produced by exhaustive enumeration of possible configurations. All other results are produced by simulating 100,000 pairs of random numbers.

The delays shown in Table 2 are module (or cell) delay. A module delay is equal to two *AND* or *OR* gate delays if two-level logic is used to implement those modules.

The propagation delays of the DIRCA, the CLA, the DICLA, and the DICLASP shown in Table 2 are plotted in Fig. 10. Two curves, $\log_2 \log_2 n$ and $2 \log_2 \log_2 n$, are also plotted in Fig. 10.

The experimental results show: First, our DICLASP has best performance even when n is small. Second, the average delay of the DICLASP is the logarithm of the logarithm of n. The constant factor, which is derived by averaging the delay of the DICLASP divided by $\log_2 \log_2 n$, is approximately 1.7. Third, the average computation time of the DICLA and the DIRCA and the worst computation time of CLA are all proportional to the logarithm of n, but the constant factor of the DIRCA is smaller than those of the DICLA and the CLA. The performance of the DIRCA is slightly better than that of the DICLA. It may be due to the overhead of computing the primary carry in the DICLA.

5 CMOS Implementation

To implement economic DICLASPs in CMOS technology, Martin's method to design economic delay-insensitive datapath circuits may be applied [18]. The behaviors of gates can be represented through a set of production rules and, then, they can be directly implemented in CMOS. Here, we just show the CMOS circuits of DIRCA and DICLASP. For more detail, see [18], [24].

TABLE 2
Program Simulation Results

Type of Adder	n-bit Adder							
	8	16	32	64	128	256	512	1024
DIRCA	3.3	4.3	5.3	6.3	7.4	8.4	9.3	10.3
CLA	5	7	9	11	13	15	17	19
DICLA	4.1	5.1	6.2	7.2	8.2	9.2	10.2	11.2
DICLASP	2.9	3.5	4.0	4.5	4.9	5.2	5.5	5.8

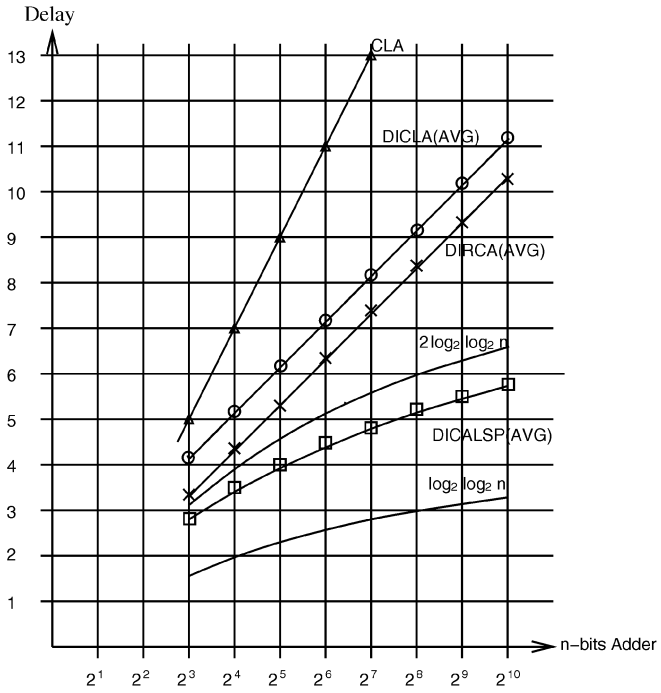


Fig. 10. Propagation delay of adders.

The CMOS implementation of DIRCA cell is shown in Fig. 11. It contains only 40 transistors. Note that, in Martin's paper [18], the transistor count per DIRCA cell is 42. The improvement is due to factoring out the subexpression as much as possible. That is, some of the transistors may be shared.

5.1 CMOS Implementation of DICLASP

The CMOS implementation of the C-module (see (14)-(18)) is shown in Fig. 12a. It contains 36 transistors. Note that k_i and g_i are implemented by static CMOS, while p_i, S_i^0, and S_i^1 are implemented by dynamic CMOS. That is, the pull-up and pull-down functions for p_i, S_i^0, and S_i^1 are not complementary. Accordingly, the output value of p_i, S_i^0, and S_i^1 may float (remain as the old value for a short time) if both pull-up and pull-down circuit are open. This causes no problem for high speed adders which take very little time to perform an addition.

The CMOS implementation of simplified D-modules (see (33)-(37)) is shown in Fig. 12b. It contains 29 transistors.

It is desirable to merge the speed-up signals into simplified D'-module so that the number of transistors needed is far smaller and the adder runs faster. The equations of simplified D'-module (see (38)-(42)) are redefined in the following way:

$$P_{i,k} = P_{i,j}P_{j-1,k} \tag{43}$$

$$K_{i,k} = P_{i,j}(k_{j-1} + \sum_{x=1}^{l-2} K_{j-1,j-2^x} + K_{j-1,k}) \tag{44}$$

$$G_{i,k} = P_{i,j}(g_{j-1} + \sum_{x=1}^{l-2} G_{j-1,j-2^x} + G_{j-1,k}) \tag{45}$$

$$C_j^0 = k_{j-1} + \sum_{x=1}^{l-2} K_{j-1,j-2^x} + K_{j-1,k} + P_{j-1,k}C_k^0 \tag{46}$$

$$C_j^1 = g_{j-1} + \sum_{x=1}^{l-2} G_{j-1,j-2^x} + G_{j-1,k} + P_{j-1,k}C_k^1. \tag{47}$$

The CMOS implementation of the simplified D'-module with speed-up circuitry is shown in Fig. 12c.

The transistor count for the simplified D'-modules with the speed-up circuitry can be derived in the following way: Consider the D'-modules at level i, where $i \geq 2$. There are 2^{k-i} D'-modules and each D'-module at level i contains $23 + 4(i-1)$ transistors Also, the speed-up circuit for the last carry needs $23 + 4k$ transistors. Thus, for an n-bit adder

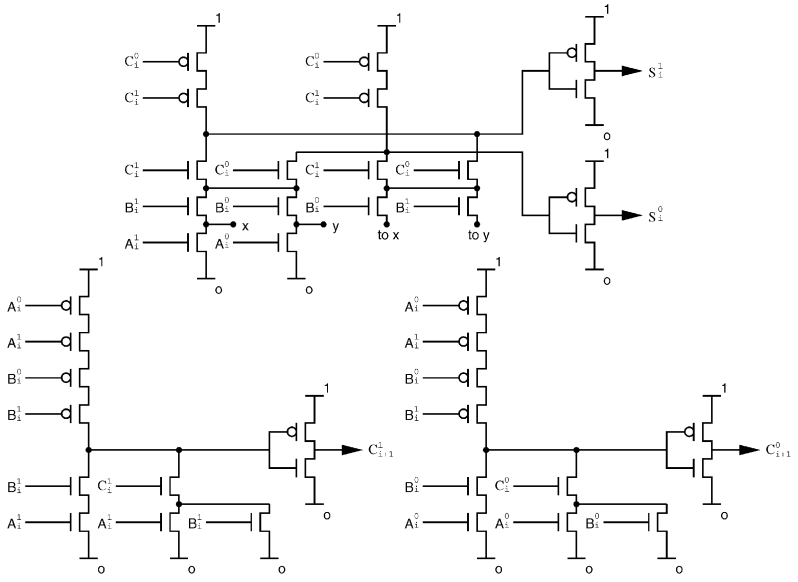


Fig. 11. CMOS implementation of DIRCA cell.

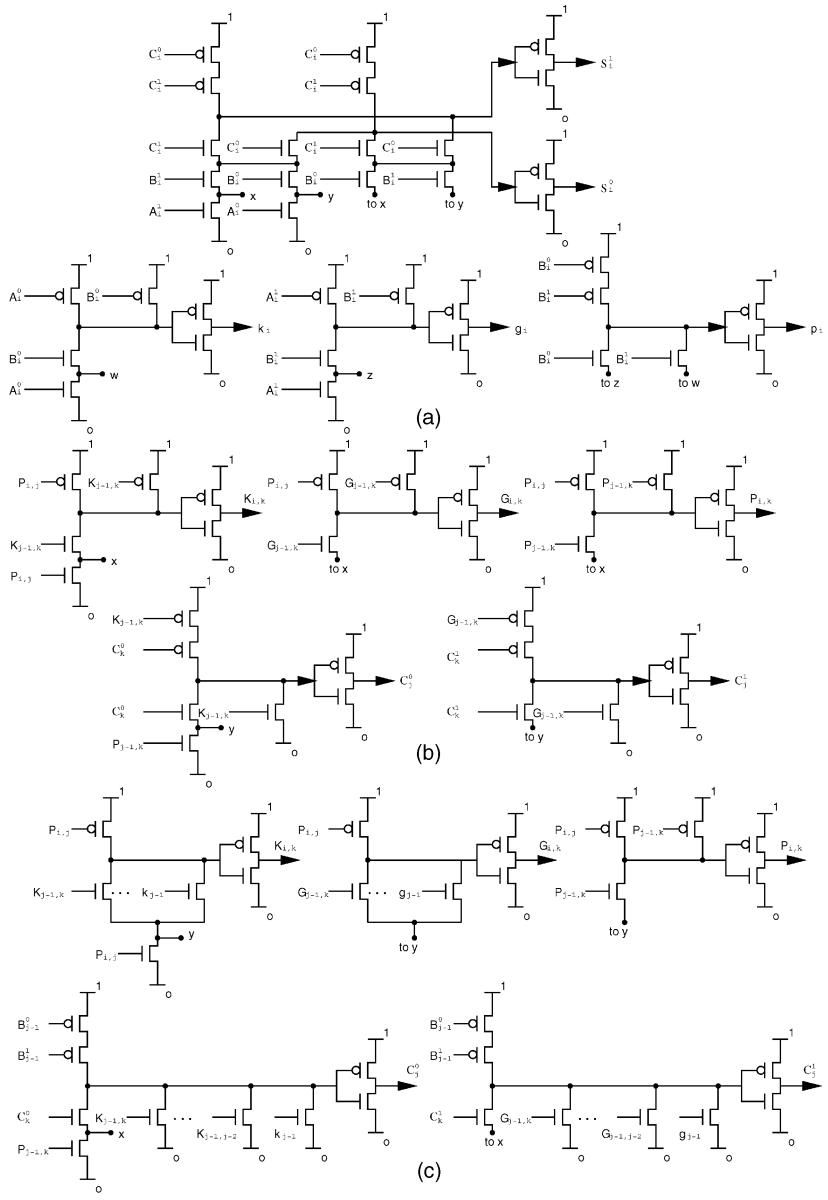


Fig. 12. CMOS Implementation of C-, SD-, and SD'-modules with speed-up circuitry. (a) CMOS implementation of C-module. (b) CMOS implementation of D-module. (c) CMOS implementation of D'-module.

$(n \geq 4)$, the speed-up circuitry needs the following transistors in total:

$$N = (23 + 4k + \sum_{i=2}^{k} 2^{k-i}(23 + 4(i-1))) = 15.5n - 4.$$

An n-bit DICLASP needs

$$36n + 29\left(\frac{n}{2}\right) + (15.5n - 4) = 66n - 4.$$

Table 3 lists the transistor counts of RCA, DIRCA, CLA, and DICLASP.

It is worth mentioning that the speed-up circuitry in fact uses a very small percentage of the CMOS area. Compared to Martin's n-bit ripple-carry adder, which needs $42n$ transistors, our n-bit DICLASP, which needs $66n - 4$ transistors, is indeed practical in high speed processors.

6 Performance Evaluation from SPICE

The SPICE simulation consists of two parts: random number inputs and statistical data gathered from a 32-bit ARM simulator. Note that our SPICE simulations are based on circuits specified only in a topological sense. Chip layouts have not been developed so that factors such as

TABLE 3
Transistor Counts of Adders

Type of Adder	Transistor Count		
	32-bit	64-bit	n-bit
RCA [18]	1280	2560	40n
DIRCA [18]	1344	2688	42n
CLA [24]	1514	3050	48n-22
DICLASP [24]	2108	4220	66n-4

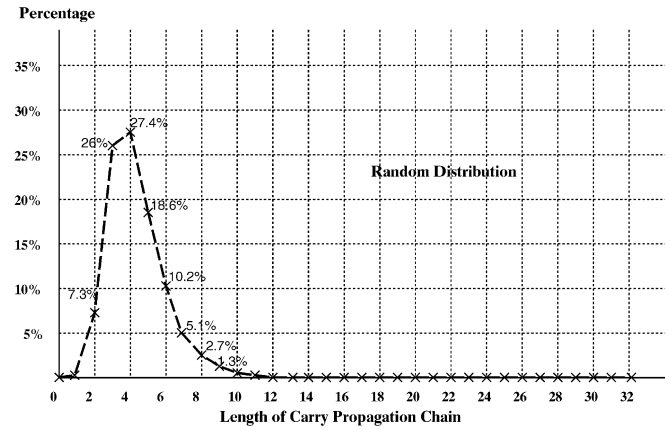


Fig. 13. Distribution of maximum propagation chains of random samples.

layout, wiring delays, stray capacitance, noise are not taken into account.

6.1 SPICE Simulation of Random Number Inputs

Ten thousand pairs of randomly generated numbers are simulated in SPICE with MOSIS 2 micron, level 2 CMOS parameters for a 32-bit DIRCA and a 32-bit DICLASP.

The propagation chain distribution of the 10,000 random samples is plotted in Fig. 13. Propagation chains with length = 2 to length = 9 take up to 98.6 percent of the sample space.

Table 4 shows the results from SPICE simulations. Note that the best (worst) case delay is the smallest (largest) delay among the simulated cases.

It is highly impractical to exhaustively simulate 32-bit self-timed adders, which would require 2^{64} cases. In our SPICE experiment, we simulate only 10,000 cases for DIRCA and DICLASP. Comparing 10,000 simulated cases to the sample space, it is obvious that only a very tiny percentage (5.42×10^{-14} percent) of cases are simulated.

Table 5 shows the confidence limits [25]: sample mean, standard deviation, confidence interval with 95 percent confidence level, and confidence interval with 99 percent confidence level for DIRCA and DICLASP. General distribution is assumed in computing confidence limits. For more detail, see [24], [25].

Statistical outcomes show that we are quite confident (99 percent) that, for random inputs, the true mean (i.e., average case performance) of a 32-bit DICLASP lies between 4.36 ns and 4.40 ns and the true mean of a 32-bit DIRCA between 6.16 ns and 6.26 ns.

The results from the random number inputs show that: First, DICLASP (DIRCA) is about 2.39 (6.34) times faster than its synchronous counterpart. Second, DICLASP is about 1.42 times faster than DIRCA. Third, the results clearly demonstrate the superiority of asynchronous circuits in the domain of average case performance vs. worst case performance of synchronous circuits. The results also contradict the report from Kinniment [26]. The report concludes that "asynchronous adders only give a performance improvement over more conventional hardware in very limited conditions," which is wrong. The major problem of Kinniment's report is that he compares CCSA, an asynchronous version of a ripple-carry adder, with a tree-like conditional-sum adder. This is unfair and misleading.

6.2 SPICE Simulation of Real Data

Based on Garside's work [3], we collected statistical data by running a 32-bit ARM emulator. The average computation times of DIRCA and DICLASP based on dynamic traces were then calculated.

Table 6 presents the statistical data obtained by simulating three benchmark programs, Dhrystone f1, Dhrystone f2, and Espresso dc2 on a 32-bit ARM emulator. All additions and subtractions performed in a 32-bit ARM ALU are analyzed and collected. The traces are categorized into four sets: Add/Subtract, Compare, Load/Store, and Branch.

A maximal carry propagation chain of inputs A and B is obtained by first XORing them, then finding the largest contiguous strings of 1s. The distribution of longest carry propagation chains is plotted in Fig. 14. The accumulated percentage is also shown for the case f1 + f2 + dc2.

The results show that: First, the distributions of Dhrystone f1 and f2 are similar and the distributions of Dhrystone and Espresso dc2 are different. This implies that different applications may have different distributions of propagation chains. Second, the distribution of f1 + f2 + dc2 is close to Espresso dc2 because it contributes 92 percent of the instructions executed. Third, from the dynamic trace of f1 + f2 + dc2, only 1.43 percent of instructions have the worst case behavior and 62.83 percent of instructions have maximal carry propagation chain less than or equal to 6. This implies that the speed-up of average case performance vs. worst case performance is significant.

It is impossible to simulate the about 1.9 million cases to calculate the average case performance of asynchronous adders. We adopt the following formula to compute average computation time of asynchronous adders.

$$ACT = \sum_{i=0}^{32} D(i) * P(i),$$

TABLE 4
Results of SPICE Simulation of Various Adders

32-bit Adders	Type of Circuit	# of Cases	Best Case Delay	Worst Case Delay	Average Case Delay
RCA	Synchronous	N/A	N/A	39.38 ns	N/A
DIRCA	Async (DI)	10,000	3.03 ns	17.4 ns	6.21 ns
CLA	Synchronous	N/A	N/A	10.46 ns	N/A
DICLASP	Async (DI)	10,000	2.69 ns	7.16 ns	4.38 ns

TABLE 5
Confidence Limits of DIRCA and DICLASP

32-bit Adder	Sample Mean	Standard Deviation	Confidence Interval (95%)	Confidence Interval (99%)
DIRCA	6.21 ns	1.897 ns	(6.17, 6.25)	(6.16, 6.26)
DICLASP	4.38 ns	0.621 ns	(4.37, 4.39)	(4.36, 4.40)

where ACT is the average computation time, $D(i)$ is the average delay for the propagation chain with length = i, and $P(i)$ is the percentage of instructions with longest propagation chain = i in a dynamic trace. The average delay for the propagation chain with length = i is computed by averaging the delays of cases where the propagation chain is in different positions. For more detail, see [24].

The performance improvement of the DI adders vs. their synchronous counterparts based on real data is shown in Table 7.

The results show that: First, the average computation times of the DIRCA and the DICLASP and based on the dynamic trace of Dhrystone f1 (f2, dc2, f1 + f2 + dc2) are 9.90 ns and 5.36 ns, ({10.03 ns and 5.40 ns}, {9.59 ns and 5.24 ns}, {9.61 ns and 5.25 ns}), respectively. Second, the average computation times of the DIRCA and the DICLASP based on the dynamic traces of Dhrystone f1 and f2 are longer than those based on Espresso dc2. This implies that the computation time is dependent on the nature of applications. Third, based on the dynamic traces, f1, f2, dc2, and f1 + f2 + dc2, DICLASP is 1.85, 1.86, 1.83, and 1.83 times faster than DIRCA, respectively. Fourth, 32-bit DICLASP and DIRCA are 1.99 and 4.10 times, on average, faster than their synchronous counterparts, respectively. Average addition time for real data is greater than for uniformly distributed random data. Nevertheless, our simulations show that our adder is substantially faster than adders operating in synchronous mode which behave as though each computation entails a worst-case (n-length) carry chain.

7 Conclusions

We have proposed a novel delay-insensitive carry-lookahead tree adder (i.e., DICLASP) in which the logic complexity is a linear function of n and the average computation time is proportional to the logarithm of the logarithm of n. To the best of our knowledge, our adder has better area-time efficiency than any other adders [8], [2], [12], [18] have.

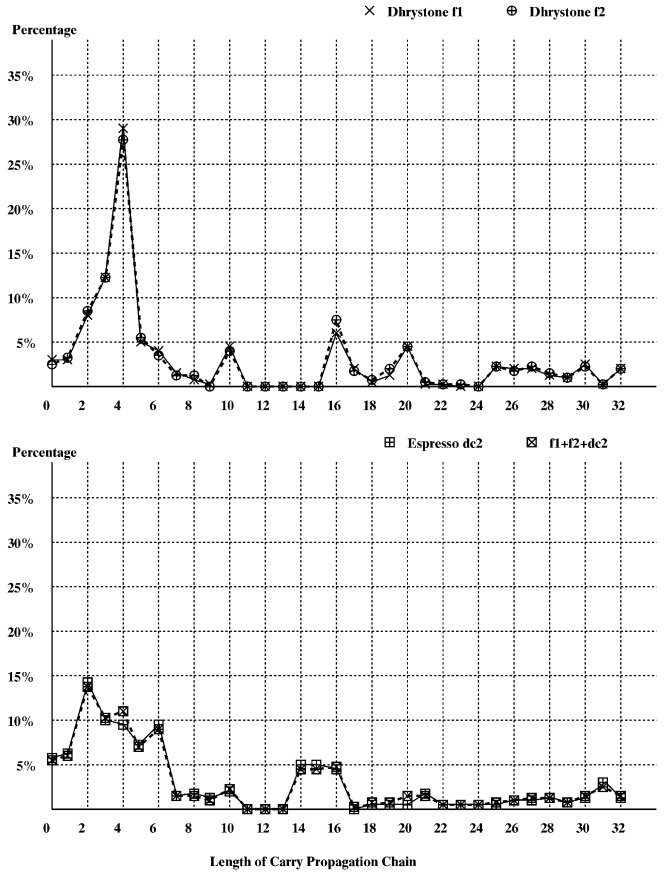


Fig. 14. Distribution of maximum propagation chain of dynamic traces.

The adder presented here is as robust as any with respect to toleration of delay variations and for no other adder is the order of computation time or the order of hardware complexity less.

The SPICE simulation results show that: First, based on random inputs, our 32-bit DICLASP is 2.39 and 1.42 times faster than its synchronous counterpart and DIRCA, repectively. Second, based on statistical data gathered from a 32-bit ARM simulator, our 32-bit DICLASP is 1.99 and 1.83 times faster than its synchronous counterpart and DIRCA, repectively.

We also present an economic CMOS implementation of our delay-insensitive carry-lookahead tree adders. The proposed adders are suitable for VLSI implementation because of their regular structure. We believe this work can be applied in the design of high speed asynchronous processors. With an interface of asynchronous and synchronous modules [27], DICLASP may be used to improve the performance of synchronous processors.

TABLE 6
Breakdown of Statistical Data from ARM Emulator

Instruction Type	Dhrystone f1		Dhrystone f2		Espresso dc2		f1+f2+dc2	
	cases	%	cases	%	cases	%	cases	%
Add/Sub	13930	15	9565	16	231519	13	255014	13
Compare	20281	22	12285	21	369157	21	401723	21
Load/Store	34657	37	22070	37	824702	46	881429	46
Branch	24279	26	15469	26	350850	20	390598	20
Total	93147		59389		1776228		1928764	

TABLE 7
Performance Improvement Based on Real Data

32-bit Adder	dynamic traces	ACT	Performance Improvement RCA (39.38 ns)	CLA (10.46 ns)
DIRCA	f1	9.90 ns	3.98	1.06
	f2	10.03 ns	3.93	1.04
	dc2	9.59 ns	4.11	1.09
	f1+f2+dc2	9.61 ns	4.09	1.09
DICLASP	f1	5.36 ns	7.35	1.95
	f2	5.40 ns	7.29	1.94
	dc2	5.24 ns	7.52	2.00
	f1+f2+dc2	5.25 ns	7.50	1.99

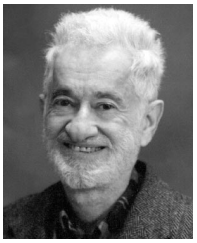
ACKNOWLEDGMENTS

The authors are grateful to Jim Garside for giving us the ARM emulator, Mark Greenstreet for proving the average computation time of CCSA is $\log n$, Steve Nowick for pointing us to Martin's work on asynchronous datapath, and anonymous referees for their critical comments that enabled us to improve the quality of this paper.

REFERENCES

[1] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.
[2] M.A. Franklin and T. Pan, "Performance Comparison of Asynchronous Adders," *Proc. Int'l Symp. Advanced Research in Asynchronous Circuits and Systems,* pp. 117-125, Nov. 1994.
[3] J.D. Garside, "A CMOS VLSI Implementation of an Asynchronous ALU," *Asynchronous Design Methodologies,* S. Furber and M. Edwards, eds., vol. A-28 of *IFIP Trans.,* pp. 181-207 1993.
[4] B. Gilchrist, J.H. Pomerene, and S.Y. Wong, "Fast Carry Logic for Digital Computers," *IRE Trans. Electronic Computers,* vol. 4, no. 4, pp. 133-136, Dec. 1955.
[5] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design.* John Wiley & Sons, 1979.
[6] B.E. Briley, "Some New Results on Average Worst Case Carry," *IEEE Trans. Computers,* vol. 22, no. 5, pp. 459-463, May 1973.
[7] A.W. Burks, H.H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," *Collected Works of John von Neumann,* vol. 5, pp. 34-79, 1963.
[8] T.-F. Ngai and M.J. Irwin, "Regular, Area-Time Efficient Carry-Lookahead Adders," *Proc. IEEE Symp. Computer Arithmetic,* pp. 9-15, 1985.
[9] I. Flores, *The Logic of Computer Arithmetic.* Prentice Hall, 1963.
[10] S.H. Unger, "Tree Realizations of Iterative Circuits," *IEEE Trans. Computers,* vol. 26, no. 4, pp. 365-393, Apr. 1977.
[11] J. Sklansky, "Conditional-Sum Addition Logic," *IRE Trans. Electronic Computers,* vol. 9, no. 6, pp. 226-231, June 1960.
[12] R.P. Brent and H.T. Kung, "A Regular Layout for Parallel Adders," *IEEE Trans. Computers,* vol. 31, no. 3, pp. 260-264, Mar. 1982.
[13] E. Brunvand, "Translating Concurrent Communicating Programs into Asynchronous Circuits," PhD thesis, Carnegie Mellon Univ., 1991.
[14] J.C. Ebergen, "A Formal Approach to Designing Delay-Insensitive Circuits," *Distributed Computing,* vol. 5, no. 3, pp. 107-119, 1991.
[15] A.J. Martin, "The Limitations to Delay-Insensitivity in Asynchronous Circuits," *Proc. Sixth MIT Conf. Advanced Research in VLSI,* William J. Dally, ed., pp. 263-278, 1990.
[16] I.E. Sutherland, "Micropipelines," *Comm. ACM,* vol. 32, no. 6, pp. 720-738, June 1989.
[17] S.H. Unger, "A Building Block Approach to Unclocked Systems," *Proc. Hawaii Int'l Conf. System Sciences,* vol. I, Jan.1993.
[18] A.J. Martin, "Asynchronous Datapaths and the Design of an Asynchronous Adder," *Formal Methods in System Design,* vol. 1, no. 1, pp. 119-137, July 1992.
[19] F.-C. Cheng, "Practical Design and Performance Evaluation of Completion Detection Circuits," *Proc. Int'l Conf. Computer Design (ICCD),* pp. 354-359, Oct. 1998.
[20] M. Greenstreet private communication, 1995.
[21] O.J. Bedrij, "Carry-Select Adder," *IRE Trans. Electronic Computers,* vol. 11, no. 6, pp. 340-346, June 1962.
[22] M. Lehman and N. Burla, "Skip Techniques for High-Speed Carry-Propagation in Binary Arithmetic Units," *IRE Trans. Electronic Computers,* vol. 10, no. 12, pp. 691-698, Dec. 1961.
[23] N.M. Martin and S.P. Hufnagel, "Conditional-Sum Early Completion Adder Logic," *IEEE Trans. Computers,* vol. 29, no. 8, pp. 753-756, Aug. 1980.
[24] F.-C. Cheng, "Synthesis of High Performance Self-Checking Delay-Insensitive Tree Circuits," PhD thesis, Dept. of Computer Science, Columbia Univ., 1998.
[25] R.L. Scheaffer and J.T. McClave, *Statistics for Engineers.* Duxbury Press, 1982.
[26] D.J. Kinniment, "An Evaluation of Asynchronous Addition," *IEEE Trans. VLSI Systems,* vol. 4, no. 1, pp. 137-140, Mar. 1996.
[27] D.S. Bormann and P.Y.K. Cheung, "Asynchronous Wrapper for Heterogeneous Systems," *Proc. Int'l Conf. Computer Design (ICCD),* Oct. 1997.

Fu-Chiung Cheng received the BS degree in computer science from Tunghai University, Taichung, Taiwan, Republic of China, in 1985 and the MS degree in computer science and engineering from Tatung University, Taipei, Taiwan, R.O.C., in 1987 and the PhD degree in computer science from Columbia University, New York, in 1998. He is an associate professor in the Department of Computer Science and Engineering, Tatung University, Taipei, Taiwan, R.O.C., and a consultant for Tatung Co., Taipei, Taiwan, R.O.C. His interests including asynchronous VLSI systems, hardware/software codesign, web-based CAD tools, and Java technology. Dr. Cheng was the recipient of the Honorable Mention Award at the 11th International Conference on VLSI Design, 1997.



Stephen H. Unger received the BEE degree from the Polytechnic Institute of Brooklyn in 1952 and the MS and ScD degrees from MIT in 1953 and 1957. He has been on the faculty of Columbia University since 1961, in the Electrical Engineering and Computer Science Department from 1961-1979 (with the rank of professor since 1968), and in the Computer Science Department since its formation in 1979. He also holds a joint appointment in the Electrical Engineering Department. Prior to coming to Columbia, he was a member of the technical staff at Bell Telephone Laboratories for just under five years. He supervised a software development group there for almost two years, after having been engaged in research on various problems in computer science. Professor Unger has been a summer and/or sabbatical leave employee of GE, IBM, RCA Laboratories, and Bell Laboratories, and a consultant for a number of companies. He was a visiting professor at the Danish Technical University in 1974-1975. He has published more than 40 technical papers and reports on topics including logic circuits, parallel processing, pattern recognition, and computer software, as well as the books *Asynchronous Sequential Switching Circuits*, *The Essence of Logic Circuits*, and *Controlling Technology: Ethics and the Responsible Engineer*. He holds one patent, is an IEEE fellow, and an AAAS fellow, and was a Guggenheim fellow in 1967. Professor Unger has also been active in the field of technology and society, is a past member of the IEEE Board of Directors, and past chair of the IEEE Ethics Committee.

Michael Theobald is a PhD student in computer science at Columbia University. He received the Diplom degree in computer science from Johann Wolfgang Goethe-Universität, Frankfurt/Main, Germany, in 1994. His research interests include synchronous and asynchronous circuits, computer-aided digital design, logic synthesis, formal verification, efficient algorithms and data structures, and combinatorial optimization. He received the Honorable Mention Award at the 1997 International Conference on VLSI Design and was a Best Paper finalist at the 1998 IEEE Async Symposium. He is a student member of the IEEE.