## CARRY LOGIC

Addition is the fundamental operation for performing digital arithmetic; subtraction, multiplication and division rely on it. How computers store numbers and perform arithmetic should be understood by the designers of digital computers. For a given weighted number system, a single digit could represent a maximum value of up to 1 less than the base or radix of the number system. A plurality of number systems exist (1). In the binary system, for instance, the maximum that each digit or bit could represent is 1. Numbers in real applications of computers are multi-bit and are stored as large collections of 16, 32, 64, or 128 bits. If the addition of multibit numbers in such a number system is considered, the addition of two legal bits could result in the production of a result that cannot fit within one bit. In such cases, a carry is said to have been generated. The generated carry needs to be added to the sum of the next two bits. This process, called carry propagation, continues from the Least Significant Bit or digit, the one that has the least weight and is the right-most, to the Most Significant Bit or digit, the one with the most weight and is the left-most. This operation is analogous to the usual manual computation with decimal numbers, where pairs of digits are added with carries being propagated towards the high-order (left) digits. Carry propagation serializes the otherwise parallel process of addition, thus slowing it down.

Since a carry can be determined only after the addition of a particular set of bits is complete, it serializes the process of multi-bit addition. If it takes a finite amount of time, say ($\Delta_g$), to calculate a carry, it will take 64 ($\Delta_g$), to calculate the carries for a 64-bit adder. Several algorithms to reduce the carry propagation overhead have been devised to speed up arithmetic addition. These algorithms are implemented using digital logic gates (2) in computers and are termed *carry logic*. However, the gains in speed afforded by these algorithms come with an additional cost which is measured in terms of the number of logic gates required to implement them.

In addition to the choice of number systems for representing numbers, they can further be represented as fixed- or floating-point (3). These representations use different algorithms to calculate a sum, although the carry propagation mechanism remains the same. Hence, throughout this article, carry propagation with respect to fixed-point binary addition will be discussed. Since a multitude of 2-input logic gates could be used to implement any algorithm, all the measurements are made in terms of the number of 2-input NAND Gates throughout this study.

### THE MECHANISM OF ADDITION

Currently, most digital computers use the binary number system to represent data. The legal digits, or bits as they are called in the binary number system, are 0 and 1. During addition, a sum, $S_i$, and a carryout, $C_i$, are produced by adding a set of bits at the $i$th position. The carry-out $C_i$ produced during the process serves as the carry-in, $C_{i-1}$, for the succeeding set of bits. Table 1 shows the underlying rules for adding two bits, $A_i$ and $B_i$, with a carry-in, $C_i$, and producing a Sum, $S_i$, and Carry-out, $C_i$.

### FULL ADDER

The logic equations that represent $S_i$ and $C_i$ of Table 1 are shown in Eq. (1) and Eq. (2). A block of logic that implements these is called full adder, and it is shown in the inset of Fig.

**Table 1. Addition of Bits $A_i$ and $B_i$ with a Carry-in $C_{i-1}$ to Produce Sum $S_i$ and Carry-out $C_i$**

| $A_i$ | $B_i$ | $C_{i-1}$ | $S_i$ | $C_i$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

1. The serial path for data through a full adder, hence its delay, is 2, as shown in Fig. 1. A full adder can be implemented using 8 gates (2) by sharing terms from Eq. (1) and Eq. (2).

$$S_i = A_iB_iC_{i-1} + \overline{A}_i\overline{B}_iC_{i-1} + \overline{A}_iB_i\overline{C}_{i-1} + A_i\overline{B}_i\overline{C}_{i-1} \quad (1)$$

$$C_i = A_iB_i + C_{i-1}(A_i + B_i) \quad (2)$$

## RIPPLE CARRY ADDER

The obvious implementation of an adder that adds two $n$-bit numbers $A$ and $B$, where $A$ is $A_nA_{n-1}A_{n-2}\ldots A_1A_0$ and $B$ is $B_nB_{n-1}B_{n-2}\ldots B_1B_0$, is a Ripple Carry Adder (RCA). By serially connecting $n$ full adders and connecting the carry-out, $C_1$, from each full adder as the $C_{i-1}$ of the succeeding full adder, it is possible to propagate the carry from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). Figure 1 shows the cascading of $n$ full adder blocks. It is clear that there is no special carry propagation mechanism in the RCA except the serial connection between the adders. Thus, the carry logic has a minimal overhead for the RCA. The number of gates required in $8n$, as each full adder is constructed with 8 gates, and there are $n$ such adders. Table 2 shows the typical gate count and speed for RCAs with varying number of bits.

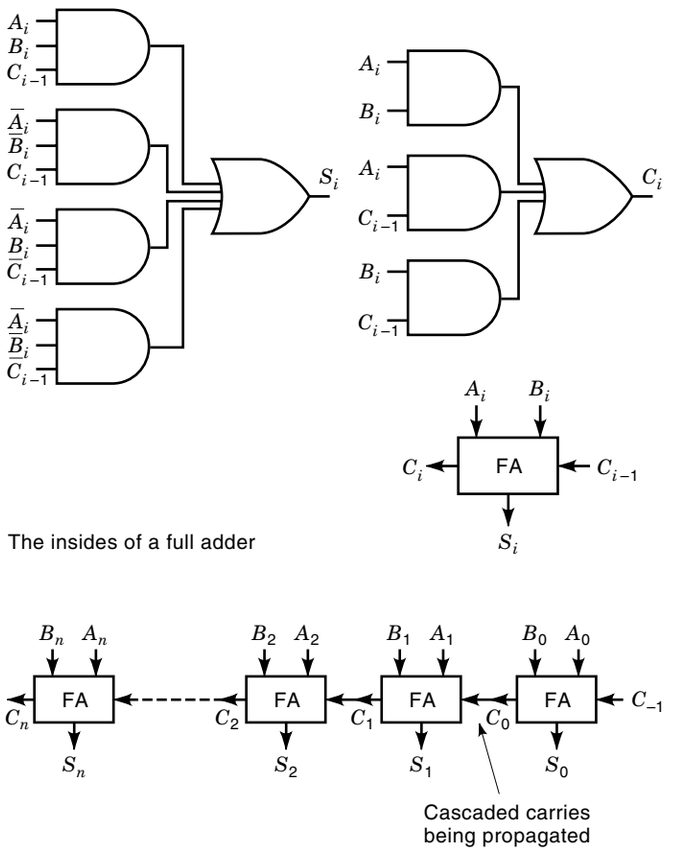


**Figure 1.** A Ripple Carry Adder ripples the carry from stage to stage using cascaded Full Adders.

**Table 2. List of Gate Counts and Delay of Various Adders**

| | Gate Count/Delay | | |
|---|---|---|---|
| Adder Type | 16-Bit | 32-Bit | 64-Bit |
| RCA | 144/36 | 288/68 | 576/132 |
| CLA | 200/10 | 401/14 | 808/14 |
| CSA | 284/14 | 597/14 | 1228/14 |
| CKA | 170/17 | 350/19 | 695/23 |

## CARRY PROPAGATION MECHANISMS

In a scenario where all the carries are available right at the beginning, addition is a parallel process. Each set of inputs $A_i$, $B_i$, and $C_{i-1}$ could be added in parallel, and the sum for 2 $n$-bit numbers could be computed with the delay of a full adder.

The input combinations of Table 1 show that if $A_i$ and $B_i$ are both 0s, then $C_i$ is always 0, irrespective of the value of $C_{i-1}$. Such a combination is called a carry kill term. For combinations where $A_i$ and $B_i$ are both 1s, $C_i$ is always 1. Such a combination is called a carry generate term. In cases where $A_i$ and $B_i$ are not equal, $C_i$ is equal to $C_{i-1}$. These are called the propagate terms. Carry propagation originates at a generate term, propagates through any successive propagate terms, and gets terminated at a carry kill or a new carry generate term. A *carry chain* is a succession of propagate terms that occur for any given input combination of $A_i$ and $B_i$. For the addition of two $n$-bit numbers, multiple generates, kills, and propagates could exist. Thus, many carry chains exist. Addition between carry chains can proceed in parallel, as there is no carry propagation necessary over carry generate or kill terms.

Based on the concept of Carry Generates, propagates, and kills, logic could be designed to predict the carries for each bit of the adder. This mechanism is static in nature. It can be readily seen that different carry chains exist for different sets of inputs. This introduces a dynamic dimension to the process of addition. The dynamic nature of the inputs could also be used and a sum computed after the carry propagation through the longest carry chain is completed. This leads to a classification into static and dynamic carry logic.

An adder that employs static carry propagation always produces a sum after a fixed amount of time, whereas the time taken to compute the sum in a dynamic adder is dependent on the inputs. In general, it is easier to design a digital system with a static adder, as digital systems are predominantly synchronous in nature, i.e., they work in lock step based on a clock that initiates each operation and uses the results after completion of a clock cycle (4).

## STATIC CARRY LOGIC

From Eq. (1), if $A_i$ and $B_i$ are both true, then $C_i$ is true. If $A_i$ or $B_i$ is true, then $C_i$ depends on $C_{i-1}$. Thus, the term $A_iB_i$ in Eq. (1) is the Generate term or $g_i$, and $A_i + B_i$ is the propagate

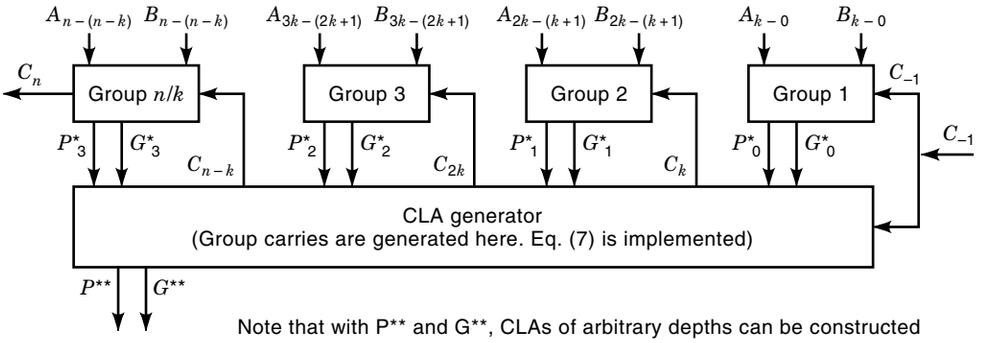


**Figure 2.** A group Carry Look Ahead Scheme with $n/k$ groups each of size $k$.

term or $p_i$. Eq. (1) can be rewritten as in Eq. (3):

$$C_i = g_i + p_i C_{i-1} \tag{3}$$

where $g_i = A_i B_i$ and $p_i = A_i + B_i$. Substituting numbers for $i$ in Eq. (3) results in Eq. (4) and Eq. (5).

$$C_1 = g_1 + p_1 C_0 \tag{4}$$

$$C_2 = g_2 + p_2 C_1 \tag{5}$$

Substituting the value of $C_1$ from Eq. (4) in Eq. (5), yields Eq. (6):

$$C_2 = g_2 + p_2 g_1 + p_2 p_1 C_0 \tag{6}$$

Generalizing Eq. (6), to any carry bit, $i$, yields Eq. (7).

$$C_i = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_1 g_1 + p_i p_{i-1} p_{i-2} \cdots p_i C_0 \tag{7}$$

By implementing logic for the appropriate value of $i$ in Eq. (7), the carry for any set of input bits can be predicted.

### Carry Look-Ahead Adder

An adder that uses Eq. (7) to generate carries for the various bits, as soon as $A$ and $B$ are available, is called a Carry Look-Ahead Adder (CLA). From Eq. (7), the carry calculation time for such an adder is 2 gate delays, and a further 2 gate delays are required to calculate the sum with bits $A_i$, $B_i$, and the generated carry. In general, for a large number of bits $n$, it is impractical to generate the carries for every bit, as the complexity of Eq. (7) increases tremendously. It is commonly practice in such cases to split the addition process into groups of $k$-bit CLA blocks that are interconnected. A group carry-lookahead adder is shown in Fig. 2. The groups now provide two new output functions $G^*$ and $P^*$, which are the group generate and propagate terms. Equation (8) and Eq. (9) provide examples of how these terms are generated for 4 bit blocks. Equation (10) shows the generation of $C_4$ using $G^*_1$ and $P^*_1$.

$$G^*_1 = g_4 + p_4 g_3 + p_4 p_3 g_2 + p_4 p_3 p_2 g_1 \tag{8}$$

$$P^*_1 = p_4 p_3 p_2 p_1 \tag{9}$$

$$C_4 = G^*_1 + P^*_1 C_0 \tag{10}$$

In typical implementations, a CLA computes the sum in $\log_2 n$ time and uses gates to the order of $n \log n$. Table 2 lists the gate count and delay of various CLAs. Thus, with some additional gate investment, considerable speed up is possible using the CLA carry logic algorithm.

Based on the CLA algorithm, several methods have been devised to speed up carry propagation even further. Three such adders that employ circuit level optimizations to achieve faster carry propagation are the Manchester Carry Adder (4), Ling Adder (5), and the Modified Ling Adder (6). However, these are specific implementations of the CLA, and do not modify carry propagation algorithms.

### Carry Select Adder

The discussion in the previous section shows that the hardware investment on CLA logic is severe. Another mechanism to extract parallelism in the addition process is to calculate two sums for each bit, one assuming a carry input of 0 and another assuming a carry input of 1, and choosing one of the sume based on the real carry generated. The idea is that the selection of one of the sums is faster than actually propagating carries through all the bits of the adder. An adder that employs this mechanism is called a Carry Select Adder (CSA) and is shown in Fig. 3. A CSA works on groups of $k$-bits, and each group works like an independent RCA. The real carry-in is always known as the LSB, and it is used as $C_0$. In Fig. 3, $C_k$ is used to select one of the sums, like $S^1_{3k-2k+1}$ or $S^0_{3k-2k+1}$ from the next group, $gp2$. In general, the selection and the addition time per bit are approximately equal. Thus, for a group that is $k$ bits wide, it approximately takes $2k$ units of time to compute the sums and a further two units of time to select the right sum, based on the actual carry. Thus, the total time for a valid carry to propagate from one group to another is $2(k+1)$ time units. Thus, for an optimal implementation, the groups in the CSA should be unequal in size, with each succeeding group being 1 bit wider than the preceding group. The gate count and speed of various CSAs is listed in Table 2.

### Carry Skip Logic

If an adder is split into groups, gp 0, gp 1 and so on of RCAs of equal width $k$, and if a carry-in of 0 is forced into each group, then the carry out from each group is its generate term. The propagate term is simple to compute and can be computed by using Eq. (9). Since the group generate terms

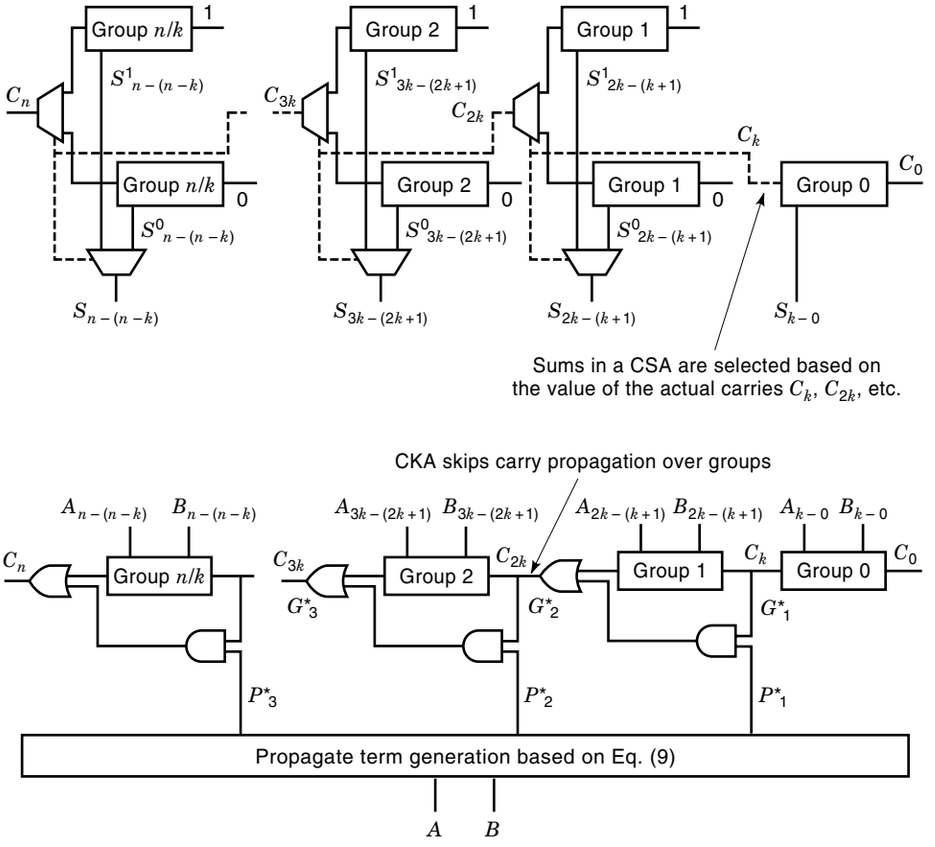


**Figure 3.** The CSA and CKA propagate carries over groups of $k$-bits.

and propagate terms are thus available, the real carry-in at each of the groups could be predicted and used to calculate the sum. An adder employing this mechanism for carry propagation is called a Carry sKip Adder (CKA) and is shown in Fig. 3. The logic gates outside of the groups in Fig. 3, implement Eq. (11), which is a generalization of Eq. (10) for the carry at any position $i$. Thus, the process of carry propagation takes place at the group level, and it is possible to skip cary propagation over groups of bits.

$$C_i = G^*_{i/k} + P^*_{i/k} C_{ki} \tag{11}$$

It takes $2k$ time units to calculate the carry from any group of size $k$. Carry propagation across groups takes an additional $n / k - 2$ time units, and it takes another $2k$ time units to calculate the final sum. Thus, the total time is $4k + n / k - 2$ time units. By making the inner blocks larger in size, it is possible to calculate the sum faster, as it is then possible to skip carry propagation over bigger groups. Table 2 lists the gate count and performance of various CKAs.

### Prefix Computation

Binary addition can be viewed as a parallel computation. By introducing an associative operator *, carry propagation and carry generation can be defined recursively. If $C_i = G_i$ in Eq. (3), then Eq. (12) with * as the concatenation operator holds. $P_i$ is the propagate term, and $G_i$ is the generate term at bit position $i$ at the boundary of a group of size $k$.

$$(G_i, P_i) = (g_i, p_i) \text{ if } i = 1 \text{ and } (g_i, p_i)^*(G_{i-1}, P_{i-1}) \quad \text{if } n \geq i > 1 \tag{12}$$

where $(g_t, p_t)^*(g_s, p_s) = (g_t + p_t g_s, p_t p_s)$ by modifying Eq. (3). Note that * is NOT commutative. All $C_i$ can be computed in parallel. Since * is associative, the recursive Eq. (12) can be broken in arbitrary ways. The logic to compute carries can be constructed recursively too. Figure 4 shows an example of carry computation using the prefix computation strategy described in Eq. (12), with block size $k = 4$ and how a combination of two 4-bit carry-logic blocks can perform 8-bit carry computation.

The CLA, CKA, CSA, and Prefix computation have been discussed in detail by Swartzlander (7), Henessey (8), and Koren (9).

## DYNAMIC CARRY LOGIC

Dynamic carry propagation mechanisms exploit the nature of the input bit patterns to speed up carry propagation and rely on the fact that the carry propagation on an average is of the order of $\log_2 n$. Due to the dynamic nature of this mechanism, valid results from addition are available at different times for different input patterns. Thus, adders that employ this technique have completion signals that flag valid results.

### Carry Completion Sensing Adder

The carry-completing sensing adder (CCA) works on the principle of creating two carry vectors, $C$ and $D$, the primary and secondary carry vectors, respectively. The 1s in $C$ are the generate terms shifted once to the left and are determined by detecting 1s in a pair of $A_i$ and $B_i$ bits, which represent the $i$th position of the addend and augend, $A$ and $B$, respectively. The 1s in $D$ are generated by checking the carries triggered by the primary carry vector $C$, and these are the propagate terms. Figure 5 shows an example for such a carry computation process. The sum can be obtained by adding $A$, $B$, $C$, and $D$ without propagating carries. A $n$-bit CCA has an approximate gate count of $17n - 1$ and a speed of $n + 4$. Hwang (10), discusses the carry-completion sensing adder in detail. Sklansky (11), provides an evaluation of several Two-summand Binary Adders.

### Carry Elimination Adder

Ignoring carry propagation, Eq. (1) describes a Half-adder, which can be implemented by a single XOR gate. In principle,

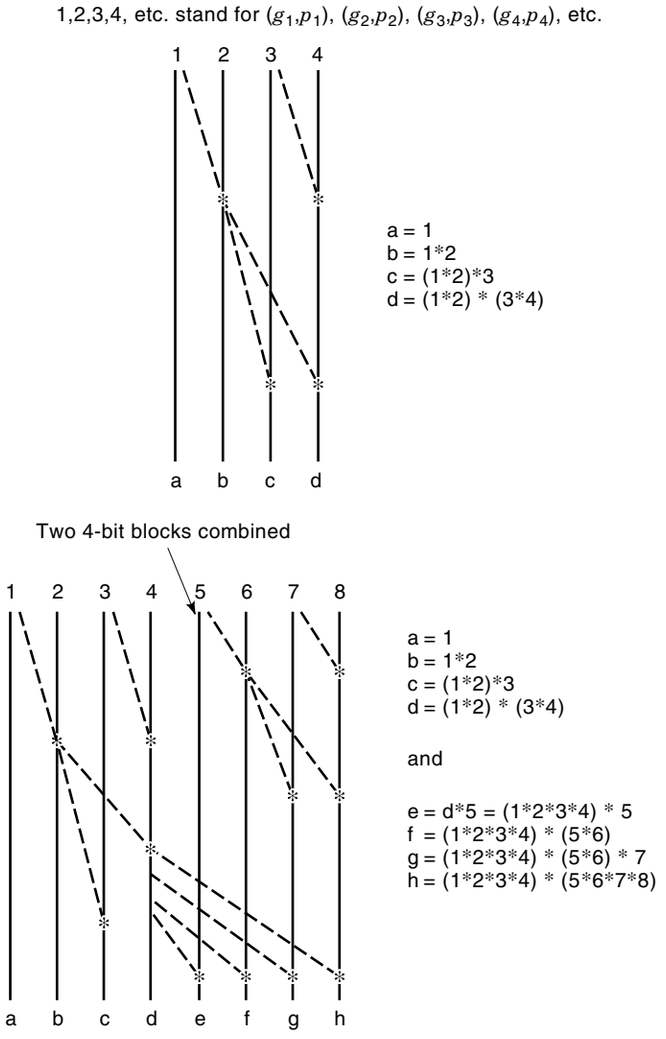


**Figure 4.** Performing 4-bit prefix computation and extending it to 8-bit numbers.

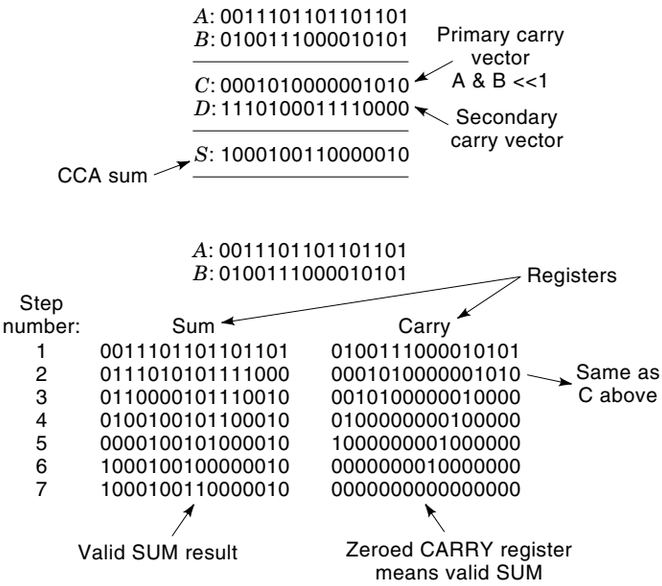


**Figure 5.** The CCA and CEA use dynamic carry propagation.

it is possible to determine the sum of 2 $n$-bit numbers by performing Half Addition on the input operands at all bit positions in parallel and by iteratively adjusting the result to account for carry propagation. This mechanism is similar to the CCA. However, the difference is that the CCA uses primary and secondary carry vectors to account for carry propagation, whereas the Carry Elimination Adder (CEA) iterates. The CEA algorithm for adding 2 numbers $A$ and $B$ is formalized by the following steps:

1. Load $A$ and $B$ into two $n$-bit storage elements called SUM and CARRY
2. Bit-Wise XOR and AND SUM and CARRY simultaneously
3. Route the XORed result back to SUM and left shift the ANDed result and route it back to the CARRY
4. Repeat the operations until the CARRY register becomes zero. At this point the result is available in SUM

The implementation of the algorithm and detailed comparisons with other carry-propagation mechanisms have been discussed by Ramachandran (12).

Figure 5 shows an example of adding two numbers using the CEA algorithm. Note that the Primary carry vector C in the CCA is the same as the CARRY register value after the first iteration. The number of iterations that the CEA performs before converging to a sum is equal to the maximum length of the carry chain for the given inputs $A$ and $B$. On average, the length of a carry chain for $n$-bit random patterns is $log_2 n$. The gate count of the CEA is about $8n + 22$ gates. It approaches the CLA in terms of speed and the CRA in terms of gate count.

## MATHEMATICAL ESTIMATION OF THE CARRY-CHAIN LENGTH

Extending from the binary number system to any $k$-ary number system, it can be said that in the addition of two $k$-ary

numbers of width $n$, ($A_{n-1}$, $A_{n-2}$, . . . $A_1$, $A_0$) and ($B_{n-1}$, $B_{n-2}$, . . ., $B_1$, $B_0$), the sum $A_i + B_i$ is:

1. In the "propagate" state if $A_i + B_i = k - 1$
2. The "generate" state if $A_i + B_i > = k$
3. The "kill" state if $A_i + B_i < k - 1$

For a given carry chain of length $j$, the probability of being in the *propagate* state is $k/k^2 = 1/k$. Define $P_n(j)$ as the probability of the addition of two uniformly random $n$-bit numbers having a maximal length carry chain of length $\geq j$.

$$P_n(j) = 0 \text{ if } n < j, \text{ and } P_n(n) = (1/k)^n \tag{13}$$

$P_n(j)$ can be computed using dynamic programming if all the outcomes contributing to probability $P_n(j)$ are split into suitable disjoint classes of events which include each contributing outcome exactly once. All outcomes contributing to $P_n(j)$ can be split into two disjoint classes of events:

Class 1: A maximal carry chain of length $\geq j$ does not start at the first position. The events of this class consist precisely of outcomes with initial prefixes having 0 up to $(j - 1)$ propagate positions followed by one nonpropagate position and then followed with the probability that a maximal carry chain of length $\geq j$ exists in the remainder of the positions. A probability expression for this class of events is shown in Eq. (14).

$$\sum_{t=0}^{j-1} \left(\frac{1}{k}\right)^t \cdot \frac{(k-1)}{k} \cdot P_{n-t-1}(t) \tag{14}$$

In Eq. (14), each term represents the condition that the first $(t + 1)$ positions are not a part of a maximal carry chain. If a position $k < t$ in some term was instead listed as nonpropagating, it would duplicate the outcomes counted by the earlier case $t = k - 1$. Thus, all outcomes with a maximal carry chain beginning after the initial carry chains of length less than $j$ are counted, and none is counted twice. None of the events contributing to $P_n(j)$ in class 1 is contained by any case of class 2 below.

Class 2: A maximal carry chain of length $\geq j$ does begin in the first position. What occurs after the end of each possible maximal carry chain beginning in the first position is of no concern. In particular, it is incorrect to rule out other maximal carry chains in the space following the initial maximal carry chain. Thus, initial carry chains of lengths $j$ through $n$ are considered. Carry chains of length less than $n$ are followed immediately by a nonpropagate position. Eq. (15) represents this condition.

$$\left(\frac{1}{k}\right)^m + \sum_{t=j}^{m-1} \left(\frac{1}{k}\right)^t \cdot \frac{(k-1)}{k} \tag{15}$$

The term $P_m(m) = (1/k)^m$ handles the case of a carry chain of full length, $m$, and the summand handles the individual cases of maximal length carry chains of length $j, j + 1, j + 2, \ldots, m - 1$. Any outcome with a maximal carry chain with length $\geq j$ not belonging to class 2 belongs to 1. In summary, any outcome with a maximal carry chain of length $\geq j$ which contributes to $P_n(j)$ is included once and only once in the disjoint collections of classes 1 and 2.

Adding the probabilities for collections 1 and 2 leads to the dynamic programming solution to $P_n(j)$ provided below, where $P_n(j) = p_n(j) + p_n(j + 1) + \cdots + p_n(n - 1) + p_n(n)$, where $p_n(i)$ is the probability of the occurrence of a maximal length carry chain of precisely length $i$. Thus, the expected value of the carry length [being the sum from $i = 1$ to $n$ of $i*p_n(i)$] becomes simply the sum of the $P_n(j)$ from $j = 1$ to $n$. Results of dynamic programming indicate that the average carry length in the *2*-ary number system for 8-bits is 2.511718750000; for 16-bits it is 3.425308227539; for 32-bits, 4.379535542335; for 64-bits, 5.356384595083; and 8.335725789691 for 128-bits.

## BIBLIOGRAPHY


1. H. L. Garner, Number Systems and Arithmetic, in F. Alt and M. Rubinoff (eds.), *Advances in Computers,* New York: Academic Press, 1965, pp. 131–194.
2. E. J. McCluskey, *Logic Design Principles,* Englewood Cliffs, NJ: Prentice-Hall, 1986.
3. S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital System Designers,* New York: Holt, Reinhart and Winston, 1982.
4. N. H. E. Weste and K. Eshragian, *Principles of CMOS VLSI Design—A Systems Perspective,* 2nd ed., Reading, MA: Addison-Wesley, 1993, Chaps. 5–8.
5. H. Ling, High Speed Binary Parallel Adder, *IEEE Trans. Comput.,* 799–802, Oct. 1966.
6. N. T. Quach and M. J. Flynn, *High-speed Addition In CMOS,* Technical Report CSL-TR-90-415, Stanford, CA: Comput. Syst. Lab., Stanford Univ., 1990.
7. E. E. Swartzlander, Jr., *Computer Arithmetic,* Washington D.C.: IEEE Computer Society Press, 1990, chapters 5–8.
8. J. L. Henessey and D. A. Patterson, *Computer Arithmetic a Quantitative Approach,* 2nd ed., California: Morgan Kauffman Publishers, A-38–A-46, 1996.
9. I. Koren, *Computer Arithmetic Algorithms,* Upper Saddle River, NJ: Prentice-Hall, 1993, pp. 7–92.
10. K. Hwang, *Computer Arithmetic,* New York: John Wiley, 1979, chap. 3.
11. J. Sklansky, An Evaluation of Several Two-Summand Binary Adders, *IRE Trans.,* **EC-9** (2): 213–226, June 1960.
12. R. Ramachandran, *Efficient Arithmetic Using Self-timing,* Masters Thesis, Corvallis, OR: Oregon State University, 1994.



RAVICHANDRAN RAMACHANDRAN
National Semiconductor Corporation

SHIH-LIEN LU
Oregon State University


**CARTESIAN COMPONENTS.** See VECTORS.