Chapter 3

Addition and Subtraction

3.1 Fixed Point Algorithms

3.1.1 Historical Review

The first electronic computers used ripple-carry addition. For this scheme, the sum at the ith bit is:

$$S_i = A_i \oplus B_i \oplus C_i,$$

where S is the sum bit, A_i and B_i are the ith bits of each operand, and C_i is the carry into the ith stage. The carry to the next stage $(i+1)$ is:

$$C_{i+1} = A_iB_i + C_i(A_i + B_i)$$

Thus, to add two n-bit operands takes at the most $n-1$ carry delays and one sum delay; but on the average the carry propagation is about $\log_2 n$ delays (see Problem 2 at the end of this chapter). In the late fifties and early sixties, most of the time required for addition was attributable to carry propagation. This observation resulted in many papers describing faster ways of propagating the carry. In reviewing these papers, some confusion may result unless one keeps in mind that there are two different approaches to speeding up addition. The first approach is *variable time addition* (asynchronous), where the objective is to detect the completion of the addition as soon as possible. The second approach is *fixed time addition* (synchronous), where the objective is to propagate the carry as fast as possible to the last stage for all operand values. Today the second approach is preferred, as most computers are synchronous, and that is the only approach we describe here. However, a good discussion of the variable time adder is given by Weigel [35] in his report "Methods of Binary Additions," which also provides one of the best overall summaries of various hardware implementations of binary adders.

Conventional fixed-time adders can be roughly categorized into two classes of algorithms: conditional sum and carry-look-ahead. Conditional sum was invented by Sklansky [22], and has been considered by Winograd [37] to be the fastest addition algorithm, but it never has become a standard integrated circuit building block. In fact, Winograd showed that with (r, d) circuits, the lower bound on addition is achievable with the conditional sum algorithm. The carry-look-ahead

$i \rightarrow$	15		14		13		12		11		10		9		8		
X_i	2		6		7		7		4		1		0		0		
Y_i	5		6		0		4		9		7		9		4		
	08	07	13	12	08	07	12	11	14	13	09	08	10	09	05	04	t_0
	083		082		082		081		139		138		095		094		t_1
	08282				08281				13895				13894				t_2
	082823895								082823894								t_3
																	t_4

$i \rightarrow$	7		6		5		4		3		2		1		0		
X_i	2		6		9		2		4		3		5		8		
Y_i	1		5		1		7		1		6		4		5		
	04	03	12	11	11	10	10	09	06	05	10	09	10	09		13	t_0
	042		041		110		109		060		059				103		t_1
	04210				04209								06003				t_2
									042096003								t_3
	08282389442096003																t_4

Selector bit = The most significant digit of each number.

The addition performed is:

$$\begin{array}{cccccccccccccccc}
2 & 6 & 7 & 7 & 4 & 1 & 0 & 0 & 2 & 6 & 9 & 2 & 4 & 3 & 5 & 8 \\
5 & 6 & 0 & 4 & 9 & 7 & 9 & 4 & 1 & 5 & 1 & 7 & 1 & 6 & 4 & 5 \\
\hline
8 & 2 & 8 & 2 & 3 & 8 & 9 & 4 & 4 & 2 & 0 & 9 & 6 & 0 & 0 & 3
\end{array}$$

At any digit position, two numbers are shown at t_0. The right number assumes no carry input, and the number on the left assumes that there is a carry input. During t_1, pairs of digits are combined, and now with each pair of digits two numbers are shown. On the right, no carry-in, and on the left, a carry-in is assumed. This process continues until the true sum results (t_4).

Figure 3.1: Example of the conditional sum mechanism.

method was first described by Weinberger and Smith in 1956 [34], and it has been implemented in standard ICs that have been used to build many different computer systems. A third algorithm described in this chapter, *canonic addition*, is a generalization of the carry-look-ahead algorithm that is faster than either conditional sum or carry-look-ahead. Canonic addition has implementation limitations, especially for long word length operands. A fourth algorithm, the Ling adder [16], uses the ability of certain circuits to perform the OR function by simply wiring together gate outputs. Ling adders provide a very fast sum, performing close to Winograd's bound, since the (r, d) circuit premise is no longer valid.

3.1.2 Conditional Sum

The principle in conditional sum is to generate, for each digit position, a sum digit and a carry digit assuming that there is a carry into that position, and another sum and carry digit assuming there is no carry input. Then pairs of conditional sums and carries are combined according to whether there is (and is not) a carry into that pair of digits. This process continues until the true sum results. Figure 3.1 illustrates this process for a decimal example.

In order to show the hardware implementation of this algorithm, the equations for a 4-bit slice can be derived.

The subscripts N and E are used to indicate no carry input and carry input (exists), respectively, to the 4-bit slice.

At each bit position the following relations hold:

$$\left.\begin{array}{lcl} S_{\mathrm{N}i} & = & A_i \oplus B_i \\ C_{\mathrm{N}(i+1)} & = & A_i B_i \end{array}\right\} \quad \text{when } C_i = 0,$$

$$\left.\begin{array}{lcl} S_{\mathrm{E}i} & = & \overline{S}_{\mathrm{N}i} \\ C_{\mathrm{E}(i+1)} & = & A_i + B_i \end{array}\right\} \quad \text{when } C_i = 1.$$

The following is a shorthand notation (which also assumes each operation takes a unit gate delay):

$$\begin{array}{rcl} G_i & = & A_i B_i \\ P_i & = & A_i + B_i \\ T_i & = & A_i \oplus B_i \quad (T_i, \text{ toggle bit}) \end{array}$$

For the 4-bit slice $i = 0, 1, 2, 3$.

$$\begin{array}{rcl} S_{\mathrm{N}0} & = & A_0 \oplus B_0 \\ S_{\mathrm{E}0} & = & \overline{S}_{\mathrm{N}0} \\ S_{\mathrm{N}1} & = & A_1 \oplus B_1 \oplus G_0 \\ S_{\mathrm{E}1} & = & A_1 \oplus B_1 \oplus P_0 \\ S_{\mathrm{N}2} & = & A_2 \oplus B_2 \oplus (G_1 + T_1 G_0) \\ S_{\mathrm{E}2} & = & A_2 \oplus B_2 \oplus (G_1 + T_1 P_0) \\ S_{\mathrm{N}3} & = & A_3 \oplus B_3 \oplus (G_2 + T_2 G_1 + T_2 T_1 G_0) \\ S_{\mathrm{E}3} & = & A_3 \oplus B_3 \oplus (G_2 + T_2 G_1 + T_2 T_1 P_0) \\ C_{\mathrm{N}4} & = & G_3 + T_3 G_2 + T_3 T_2 G_1 + T_3 T_2 T_1 G_0 \\ C_{\mathrm{E}4} & = & G_3 + T_3 G_2 + T_3 T_2 G_1 + T_3 T_2 T_1 P_0 \end{array}$$

Of course, terms such as $G_1 + T_1 G_0$ could also be written in the more familiar form $G_1 + P_1 G_0$, which is logically equivalent. Replacing T_i with P_i may simplify the implementation.

Thus, the 4-bit sums are generated, and the true sum is selected according to the lower order carry-in, i.e:

$$\begin{array}{l} S_0 = S_{\mathrm{E}0} C_0 + S_{\mathrm{N}0} \overline{C}_0 \\ \vdots \quad\;\; \vdots \quad\quad\;\; \vdots \\ S_3 = S_{\mathrm{E}3} C_0 + S_{\mathrm{N}3} \overline{C}_0 \end{array}$$

Figure 3.2 shows the logic diagram of a 4-bit slice (conditional sum) adder.

In general, the true carry into a group is formed from the carries of the previous groups. In order to speed up the propagation of the carry to the last stage, look-ahead techniques can be derived assuming a 4-bit adder as a basic block. The carry-out of bit $i(C_i)$ is valid whenever a carry-out is developed within the 4-bit group $(C_{\mathrm{N}i})$, or whenever there is a conditional carry-out $(C_{\mathrm{E}i})$ for the group *and* there was a valid carry-in (C_{i-4}). Using this, we have:

$$C_4 \quad = \quad C_{\mathrm{N}4} + C_{\mathrm{E}4} C_0$$

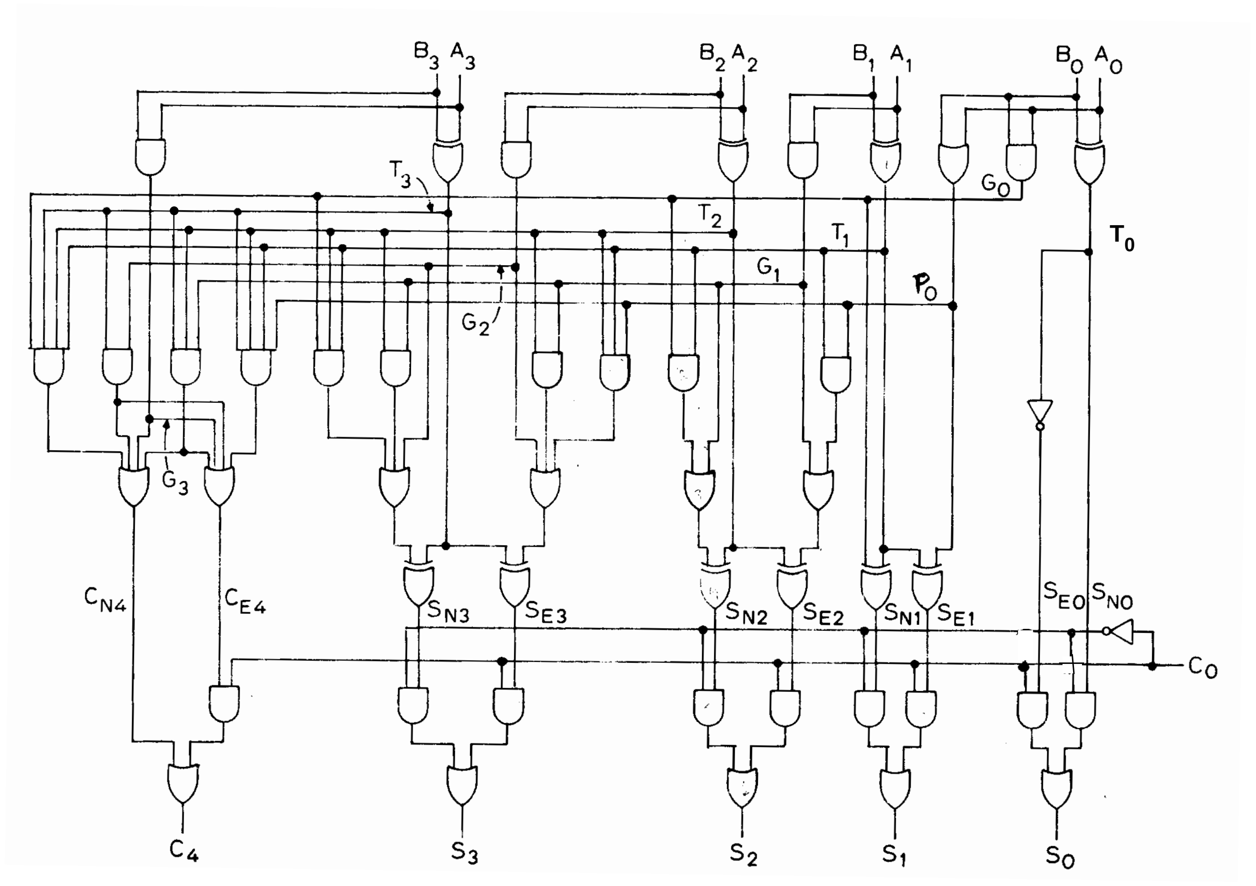


Figure 3.2: 4-bit conditional sum adder slice with carry-look-ahead (gate count= 45).

$$
\begin{aligned}
C_8 &= C_{N8} + C_{E8}C_4 \\
C_8 &= C_{N8} + C_{E8}C_{N4} + C_{E8}C_{E4}C_0 \\
C_{12} &= C_{N12} + C_{E12}C_8 \\
C_{12} &= C_{N12} + C_{E12}C_{N8} + C_{E12}C_{E8}C_{N4} + C_{E12}C_{E8}C_{E4}C_0 \\
C_{16} &= C_{N16} + C_{E16}C_{12} \\
C_{16} &= C_{N16} + C_{E16}C_{N12} + C_{E16}C_{E12}C_{N8} + C_{E16}C_{E12}C_{E8}C_{N4} + \\
&\quad C_{E16}C_{E12}C_{E8}C_{E4}C_0
\end{aligned}
$$

Note that a fan-in of 5 is needed in the preceding equations to propagate the carry across 16 bits in two gate delays. Thus, 16-bit addition can be completed in seven gate delays: three to generate conditional carry, two to propagate the carry, and two to select the correct sum bit. This delay can be generalized for n bits and r fan-in (for $r \geq 4$ and $n \geq r$) as:

$$t = 5 + 2\left\lceil \log_{r-1}(\lceil n/r \rceil - 1) \right\rceil \tag{3.1}$$

The factor 5 is determined by the longest C_4 path in Figure 3.2. The n bits of each operand are broken into $\lceil \frac{n}{r} \rceil$ groups, as shown in the equations for C_{N4} and C_{E4}, but since the lowest order

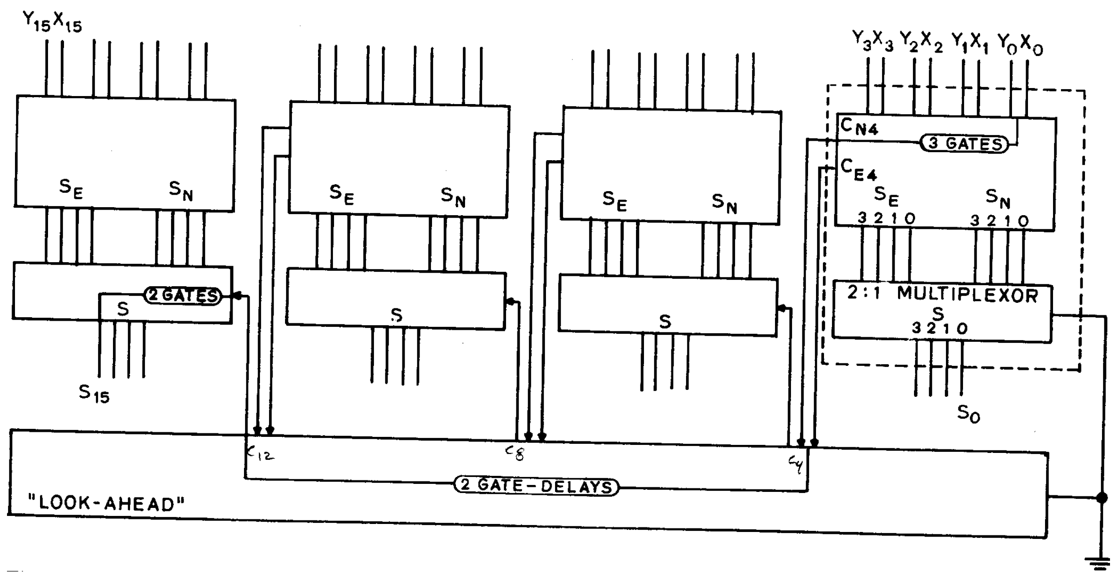


Figure 3.3: 16-bit conditional sum adder. The dotted line encloses a 4-bit slice with internal look ahead. The rectangular box (on the bottom) accepts conditional carries and generates fast true carries between slices. The worst case path delay is seven gates.

group (C_4) is already known, only $\lceil \frac{n}{r} \rceil - 1$ groups must be resolved. Finally, sum resolution can be performed on $r - 1$ groups per AND–OR gate pair (see preceding equation for C_{16}), with two delays for each pair.

If $r \gg 1$, then $r \simeq r - 1$, and if $r \ll n$, then

$$t \simeq 3 + 2\lceil \log_r n \rceil$$

The delay equation (3.1) is correct for $r \geq 4$. For $r = 3$ or $r = 2$ and $n \leq r$, then $t = 7$. The cases $r = 3$ and $r = 2$ where $n > r$ are left as an exercise.

3.1.3 Carry-Look-Ahead Addition

In the last decade, the carry-look-ahead has become the most popular method of addition, due to a simplicity and modularity that make it particularly adaptable to integrated circuit implementation. To see this modularity, we derive the equations for a 4-bit slice.

The sum equations for each bit position are:

$$\left.\begin{aligned} S_0 &= A_0 \oplus B_0 \oplus C_0 \\ S_1 &= A_1 \oplus B_1 \oplus C_1 \\ S_2 &= A_2 \oplus B_2 \oplus C_2 \\ S_3 &= A_3 \oplus B_3 \oplus C_3 \end{aligned}\right\} \quad \begin{aligned} &\text{in general:} \\ &S_i = A_i \oplus B_i \oplus C_i \end{aligned}$$

The carry equations are as follows:

$$\left.\begin{aligned} C_1 &= A_0B_0 + C_0(A_0 + B_0) \\ C_2 &= A_1B_1 + C_1(A_1 + B_1) \\ C_3 &= A_2B_2 + C_2(A_2 + B_2) \\ C_4 &= A_3B_3 + C_3(A_3 + B_3) \end{aligned}\right\} \quad \begin{array}{l} \text{in general:} \\ C_{i+1} = A_iB_i + C_i(A_i + B_i) \end{array}$$

The general equations for the carry can be verbalized as follows: there is a carry into the $(i+1)$th stage if a carry is generated locally at the ith stage, or if a carry is propagated through the ith stage from the $(i-1)$th stage. Carry is generated locally if both A_i and B_i are ones, and it is expressed by the generate equation $G_i = A_iB_i$. A carry is propagated only if either A_i or B_i is one, and the equation for the propagate term is $P_i = A_i + B_i$.
We now proceed to derive the carry equations, and show that they are functions only of the previous generate and propagate terms:

$$\begin{aligned} C_1 &= G_0 + P_0C_0 \\ C_2 &= G_1 + P_1C_1 \end{aligned}$$

Substitute C_1 into the C_2 equation (in general, substitute C_i in the C_{i+1} equation):

$$\begin{aligned} C_2 &= G_1 + P_1G_0 + P_1P_0C_0 \\ C_3 &= G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\ C_4 &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{aligned}$$

We can now generalize the carry-look-ahead equation:

$$C_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \cdots + P_iP_{i-1}\ldots P_0C_0$$

This equation implies that a carry to any bit position could be computed in two gate delays, if it were not limited by fan-in and modularity; but the fan-in is a serious limitation, since for an n-bit look ahead the required fan-in is n, and modularity requires a somewhat regular implementation structure so that similar parts can be used to build adders of differing operand sizes. This modularity requirement is what distinguishes the CLA algorithm from the canonic algorithm discussed in the next section.

The solution of the fan-in and modularity problems is to have several levels of carry-look-ahead. This concept is illustrated by rewriting the equation for C_4 (assuming fan-in of 4, or 5 if a C_0 term is required):

$$C_4 = \underbrace{G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0}_{\text{Group generate } = G'_0} + \underbrace{P_3P_2P_1P_0}_{\text{Group propagate } = P'_0} C_0$$

$$C_4 = G'_0 + P'_0C_0$$

Notice the similarity of C_4 in the last equation to C_1. Similarly, the equations for C_5 and C_6 resemble those for C_2 and C_3.

The CLA level consists of the logic to form fan-in limited generate and progate terms. It requires two gate delays. With a fan-in of 4, two levels of carry-look-ahead (CLA) are sufficient for 16

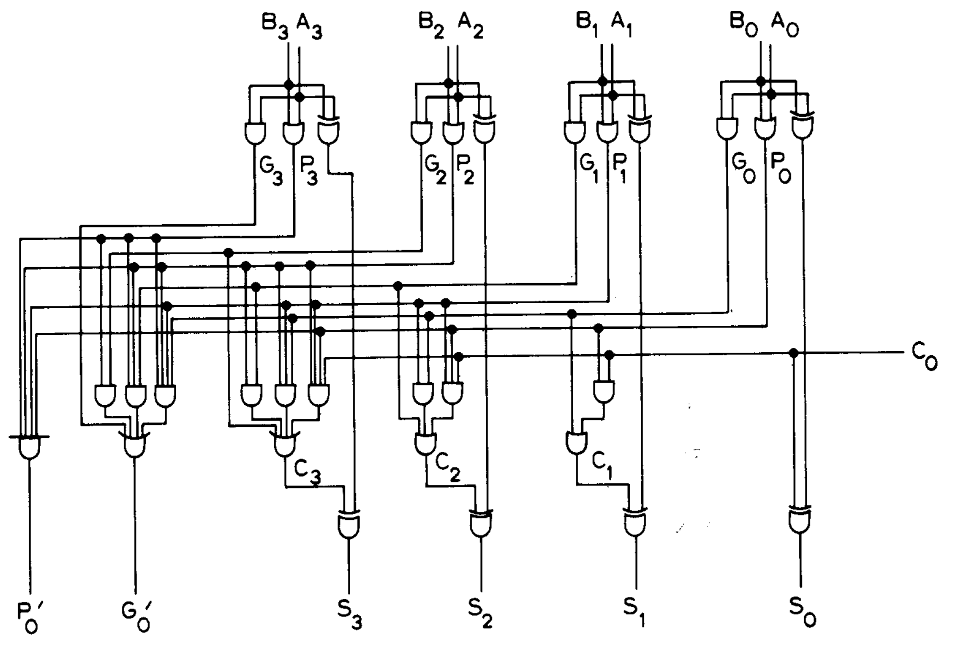


Figure 3.4: 4-bit adder slice with internal carry-look-ahead (gate count = 30).

bit additions. Similarly, CLA of between 17 and 64 bits requires a third level. In general, CLA across 17 to 64 bits requires a second level of carry generator. In general, the number of CLA levels is:

$$\lceil \log_r n \rceil$$

where r is the fan-in, and n is the number of bits to be added.

We now describe the hardware implementation of a carry-look-ahead addition. It is assumed that the fan-in is 4; consequently, the building blocks are 4-bit slices. Two building blocks are necessary. The first one is a 4-bit adder with internal carry-look-ahead across 4 bits, and the second one is 4 group carry generator. Figure 3.4 shows the gate level implementation of the 4-bit CLA adder, according to the equations for S_0 through S_3 and C_1 through C_3.

Figure 3.5 is the gate implementation of the four group CLA generator. The equations for this generator are as follows, where G'_0 and P'_0 are the (0–3) *group* generate and propagate terms (to distinguish them from G_0 and P_0, which are *bit* generate and propagate terms):

$$\begin{aligned} C_4 &= G'_0 + P'_0 C_0 \\ C_8 &= G'_1 + P'_1 G'_0 + P'_1 P'_0 C_0 \\ C_{12} &= G'_2 + P'_2 G'_1 + P'_2 P'_1 G'_0 + P'_2 P'_1 P'_0 C_0 \end{aligned}$$

and the third level generate (G'') and propagate (P'') terms are:

$$\begin{aligned} G'' &= G'_3 + P'_3 G'_2 + P'_3 P'_2 G'_1 + P'_3 P'_2 P'_1 G'_0 \\ P'' &= P'_3 P'_2 P'_1 P'_0 \end{aligned}$$

The G'' and P'' are more completely labeled G''_0 and P''_0. The corresponding third level carrys

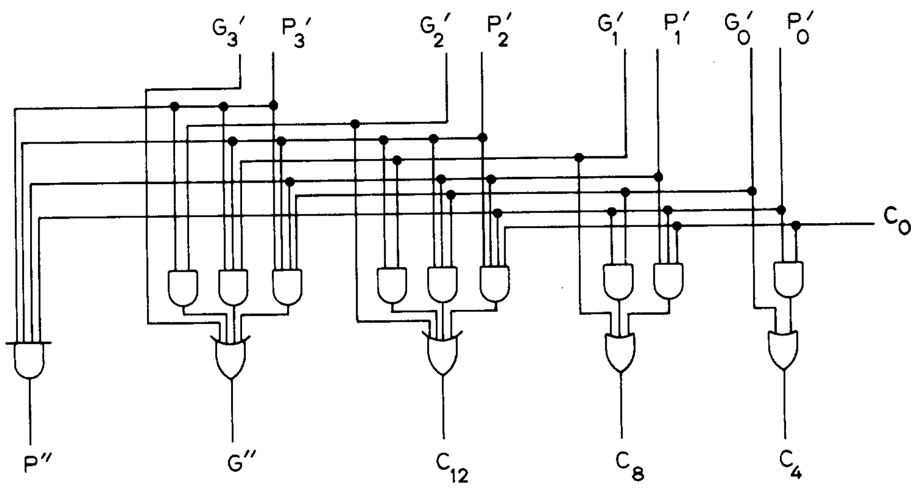


Figure 3.5: Four group carry-look-ahead generator (gate count = 14).

are:

$$\begin{aligned} C_{16} &= G_0'' + P_0''C_0 \quad (C_0 \text{ can be end around carry}) \\ C_{32} &= G_1'' + P_1''G_0'' + P_1''P_0''C_0 \\ C_{48} &= G_2'' + P_2''G_1'' + P_2''P_1''G_0'' + P_2''P_1''P_0''C_0 \\ C_{64} &= G_3'' + P_3''G_2'' + P_3''P_2''G_1'' + P_3''P_2''P_1''G_0'' + P_3''P_2''P_1''C_0 \end{aligned}$$

The implementation of a 64-bit addition from these building blocks is shown in Figure 3.6. From this figure, we can derive the general equation for worst case path delay (in gates) as a function of fan-in and number of bits.

The longest path in the 64-bit addition consists of the following delays:

• Initial generate term	per bit	1 gate delay
• Generate term	across 4 bits	2 gate delays
• Generate term	across 16 bits	2 gate delays
• C_{48} generation		2 gate delays
• C_{60} generation		2 gate delays
• C_{63} generation		2 gate delays
• S_{63} generation		1 gate delays
	Total =	12 gate delays

In general, for n-bit addition limited by fan-in of r:

• Generate term	per bit	1 gate delay
• Generate C_n	2×(2(number of CLA levels)−1) gate delays	
• Generate S_n		1 gate delay

Total CLA gate delays = 2 + 4 (number of CLA levels) − 2

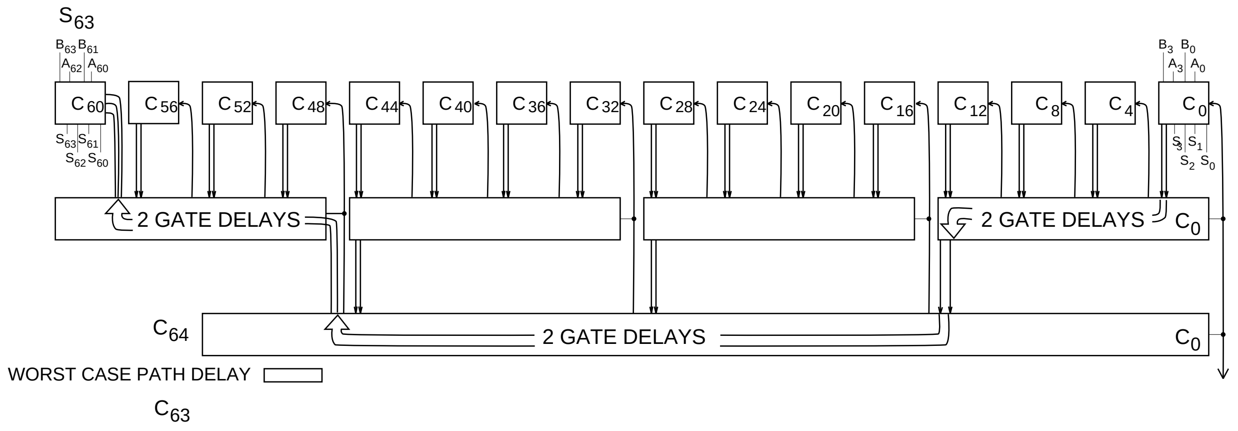


Figure 3.6: 64-bit addition using full carry-look-ahead. The first row is made of a 4-bit adder slice with internal carry-look-ahead (see Figure 3.4). The rest are look ahead carry generators (see Figure 3.5). The worst case path delay is 12 gates (the delay path is strictly for addition).

$$\text{Total CLA gate delays} = 4\ (\text{number of CLA levels}).$$

The number of CLA levels is $\lceil \log_r n \rceil$.

$$\boxed{\text{CLA gate delays} = 4\lceil \log_r n \rceil}$$

Before we conclude the discussion on carry-look-ahead, it is interesting to survey the actual integrated circuit implementations of the adder-slice and the carry-look-ahead generator. The TTL 74181 [30] is a 4-bit slice that can perform addition, subtraction, and several Boolean operations such as AND, OR, XOR, etc. Therefore, it is called an ALU (Arithmetic Logic Unit) slice. The slice depicted in Figure 3.4 is a subset of the 74181. The 74182 [30] is a four-group carry-look-ahead generator that is very similar in implementation to Figure 3.5. The only difference is in the opposite polarity of the carries, due to an additional buffer on the input carry. (Inspection of Figure 3.6 shows that the Generate and Propagate signals drive only one package, regardless of the number of levels, whereas the carries' driving requirement increases directly with the number of levels.) For more details on integrated circuit implementation of adders, see Waser [32].

3.1.4 Canonic Addition: Very Fast Addition and Incrementation

So far, we have examined the delay in practical implementation algorithms—conditional sum and CLA—as well as reviewing Winograd's theoretic delay limit. Now Winograd [36] shows that his bound of binary addition is achievable using (r, d) circuits with a conditional sum algorithm. The question remaining is what is the fastest known binary addition algorithm using conventional AND–OR circuits (fan-in limited without use of a wired OR).

Before developing such fast adders, called canonic adders, consider the problem of incrementation—simply adding one to X, an n-bit binary number. Winograd's approach would yield a bound on an increment of:

$$\text{Increment } (r,d) \text{ delays} = \lceil \log_r (n+1) \rceil .$$

Such a bound is largely realizable by AND circuits, since the longest path delay (the highest order sum bit, S_n) depends simply on the configuration of the old value of X. Thus, if we designate I as the increment function:

$$\begin{array}{rr} & X_{n-1}X_{n-2}\ldots X_0 \\ + & I \\ \hline C_n S_{n-1} & S_{n-2}\ldots S_0 \end{array} .$$

Then C_n, the overflow carry, is determined by:

$$C_n = \prod_{i=0}^{n-1} X_i \cdot I \quad \text{(i.e., the AND of all elements in X)},$$

and intermediate carries, C_j:

$$C_j = \prod_{i=0}^{j-1} X_i \cdot I .$$

C_n is implementable as a fan-in limited tree of AND circuits in:

$$C_n \text{ gate delays } = \lceil \log_r (n+1) \rceil .$$

Each output S_j bit in the increment would have an AND tree:

$$\begin{aligned} S_0 &= X_0 \oplus I \\ S_j &= X_j \oplus C_j \end{aligned}$$

Thus, the delay in realizing S_{n-1} (the nth sum bit) is:

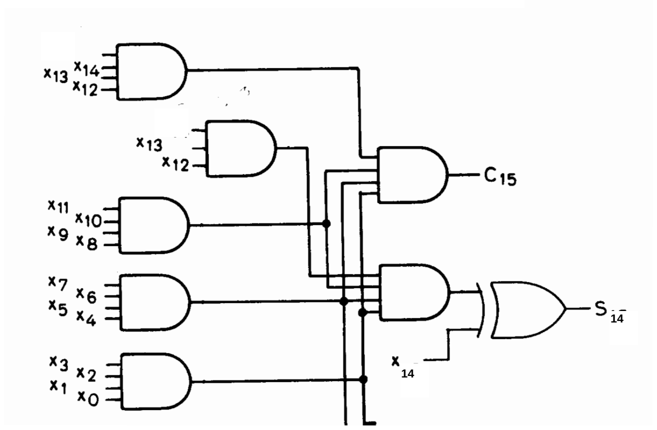
$$\text{Increment gate delays } = \lceil \log_r n \rceil + 1,$$

that is, the gate delays in C_{n-1} plus the final exclusive OR.

Example 3.1

A 15-bit incrementer (bits 0–14) might have the following high order configuration for S_{14} and

C_{15}.



The amount of hardware required to implement this approach is not as significant as it first appears. The carry-out circuitry requires:

$$\left\lceil \frac{n}{r} \right\rceil + \left\lceil \frac{n}{r} \cdot \frac{1}{r} \right\rceil + \ldots \text{ gates,}$$

or approximately

$$\left\lceil \frac{n}{r} \right\rceil \left(1 + \frac{1}{r} + \frac{1}{r^2} \cdots \right),$$

where the series consists of only a few terms, as it terminates for the lowest k that satisfies:

$$\left\lfloor \frac{n}{r^k} \right\rfloor = 1.$$

The familiar geometric series $(1 + \frac{1}{r} + \frac{1}{r^2} + \cdots)$ can conservatively be replaced by its infinite sum $\frac{r}{r-1}$. Thus:

$$\text{number of increment gates in } C_n \leq \left\lceil \frac{n}{r} \right\rceil \left(\frac{r}{r-1}\right)$$

and summing over the carry terms and adding the n exclusive ORs for the sums,

$$\text{total increment gates } \leq \sum_{i=1}^{n} \left\lceil \frac{i}{r} \right\rceil \left(\frac{r}{r-1}\right) + n$$

or, ignoring effects of the ceiling,

$$\text{total increment gates } \simeq \frac{n(n+1)}{2(r-1)} + n.$$

$\diamondsuit$

Now, most of these gates can be shared by lower order sum terms (fan-out permitting). Thus, for lower order terms (e.g., S_{n-2}):

$$S_{n-2} = (X_{n-2} \cdot X_{n-3} \cdots X_{n-2-r}) \cdot (\text{existing terms from } C_{n-2-r}).$$

Thus, only two additional circuits per lower order sum bit are required. The total number of increment gates is then approximately:

$$\text{increment gates } \simeq \left\lceil \frac{n}{r} \right\rceil + 2(n-1) + n \simeq \left\lceil \frac{n}{r} \right\rceil + 3n,$$

e.g., for $r = 4$ and $n = 32$ the total number of gates is 104 gates.

The same technique can be applied to the problem of n-bit binary addition. Here, in order to add two n-bit binary numbers:

$$\begin{array}{r} X_{n-1}X_{n-2}\ldots X_0 \\ +Y_{n-1}\,Y_{n-2}\ldots Y_0 \\ \hline S_{n-1}\,S_{n-2}\ldots S_0 \end{array}$$

We have:

$$\begin{array}{lcl} C_n = G_{n-1} & \text{designated as} & C_n = C_n^{n-1} \\ \quad +P_{n-1}\cdot G_{n-2} & & \quad +C_n^{n-2} \\ \quad +P_{n-1}\cdot P_{n-2}\cdot G_{n-3} & & \quad +C_n^{n-3} \\ \quad + & & \quad + \\ \quad \vdots & & \quad \vdots \\ \quad +\prod_{i=1}^{n-1} P_i \cdot G_0 & & \quad +C_n^0 \end{array},$$

and for each sum bit S_j $(n-1 \geq j > 0)$,

$$\begin{array}{rl} & S_j = X_j \oplus Y_j \oplus C_j \\ \text{and} & S_0 = X_0 \oplus Y_0 \end{array}$$

In the above, $G_i = X_i \cdot Y_i$, $P_i = X_i + Y_i$ and C_n^i designates the term that generates a carry-out of bit i and propagates it to a bit n. This is simply an AND–OR expansion of the required carry—hence the term "canonic addition."

The C_n term consists of an n-way OR, the longest of whose input paths is an n-way AND which generates in bit 0 and propagates elsewhere. Note that since $G_i = X_i \cdot Y_i$, a separate level to form G_i is not required, but each P_i requires an OR level.

Thus, the number of gate delays is $\lceil \log_r n \rceil$ for the AND tree and a similar number for the OR tree, plus one for the initial P_i:

$$\text{Gate delays in } C_n = 2\lceil \log_r n \rceil + 1.$$

The formation of the highest order sum (S_{n-1}) requires the formation of C_{n-1} and a final exclusive OR. Thus,

$$\text{Gate delays in } \mathrm{S}_n = 2\lceil \log_r (n-1) \rceil + 2.$$

Actually, the delay bound can be slightly improved in a number of cases by arranging the inputs to the OR tree so that short paths such as G_{n-1} or $P_{n-1} \cdot G_{n-1}$ are assigned to higher nodal inputs, while long paths such as $\prod_{i=k}^{n-1} P_k \cdot G_k$ $(k = 1, 2, 3 \ldots)$ are assigned to a lower node.

This prioritization of the inputs to the OR tree provides a benefit in a number of cases where the number of inputs n exceeds an integer tree boundary by a limited amount. The AND terms use from one gate delay to $\lceil \log_r n \rceil$ gate delays. If we can site the slow AND terms in fast positions in the OR tree (and there are enough of them!), we can save a gate delay ($\delta = 1$). For example, if $r = 4$ and $n = 7$, we would have

$$C_7 = G_6 + P_6 \cdot G_5 + P_6P_5G_4 + P_6P_5P_4G_3 + P_6P_5P_4P_3G_2 + P_6P_5P_4P_3P_2G_1 + P_6P_5P_4P_3P_2P_1G_0$$

The first four AND terms are generated in one gate delay, while the remaining three terms require two delays ($r = 4$). However, the OR tree consists of seven input terms—four at the second level and three at the root. Thus, the slow AND terms can be accommodated in the three fast (first-level) sites in the OR tree, saving a gate delay.

More generally, the number of long path terms in the AND tree is

$$n - r^{\lceil \log_r n \rceil - 1}.$$

The OR (with $\lceil \log_r n \rceil$ levels) has

$$r^{\lceil \log_r n \rceil - 1},$$

total preferred sites of which

$$\left\lceil \frac{n - r^{\lceil \log_r n \rceil - 1}}{r - 1} \right\rceil$$

have been used for the highest level inputs. Thus,

$$r^{\lceil \log_r n \rceil - 1} - \left\lceil \frac{n - r^{\lceil \log_r n \rceil - 1}}{r - 1} \right\rceil$$

are the number of preferred sites available in the OR tree. Now, if the available site equals or exceeds the number of long AND paths, we have a savings of one gate delay:

$$n - r^{\lceil \log_r n \rceil - 1} \leq r^{\lceil \log_r n \rceil - 1} - \lceil \log_r (n - r^{\lceil \log_r n \rceil - 1} \rceil$$
$$n \leq 2r^{\lceil \log_r n \rceil - 1} - \lceil \log_r (n - r^{\lceil \log_r n \rceil - 1}) \rceil$$

Thus, the exact delay is:

$$\boxed{\text{Canonic addition gate delays} = 2\lceil \log_r (n-1) \rceil + 2 - \delta}$$

where δ is the Kronecker δ and is equal to 1 whenever $\lceil \log_r n \rceil > 1$ and the above integer boundary condition is satisfied.

Consider the example $r = 4$ and $n = 20$.

Now $\lceil \log_4 20 \rceil = 3$

and $r^{\lceil \log_r n \rceil - 1} = r^{3-1} = 16$

and $\log_4 (20 - 4^2) = 1$

Since $n = 20 \leq 32 - 1$

then $\delta = 1$

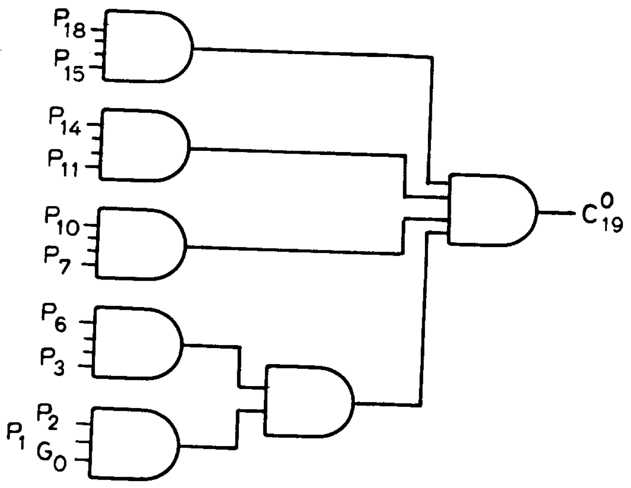
$$\text{Gate delays} = 2\lceil \log_4 19 \rceil + 2 - 1$$
$$= 7$$

Whereas

$$\text{Winograd's bound} = \lceil \log_4 2 \cdot 20 \rceil = \lceil \log_4 40 \rceil, \\ = 3$$

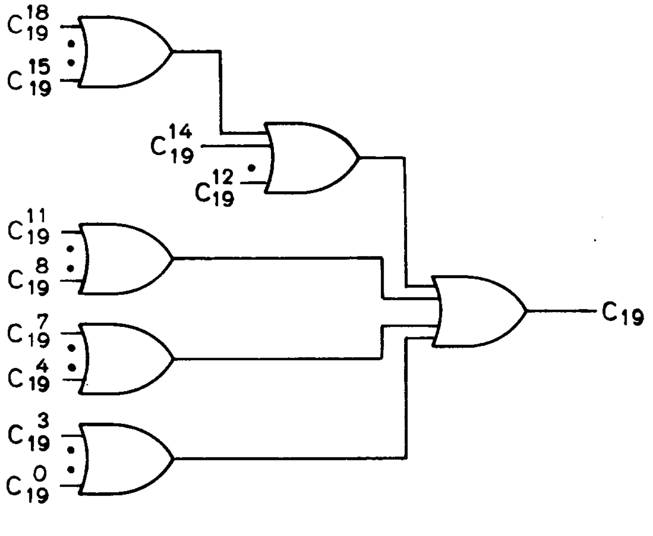
Example 3.2

The AND tree for the *generate* in bit 0 and propagate to bit 19 (C_{19}^0) is:



Terms C_{19}^1, C_{19}^2 and C_{19}^3 (two stages of delay) will have similar structures (i.e., three stages of delay), however, lower stages C_{19}^i for i between 4 and 14 have two stages of delay, while C_{19}^{15} through G_{19} have one stage. ◇

Thus, in the OR tree we must insure that terms C_{19}^0 through C_{19}^3 are preferentially situated ($\delta = 1$).



The amount of hardware required is not determinable in a straightforward way, especially for the AND networks. For the OR networks, we have:

4 bits at 6 gates
4 bits at 5 gates
8 bits at 4 gates'
4 bits at 1 gate

or 80 gates total. To this must be added 20×2 gates for initial propagate and generate terms. The AND gates required for bit 19 include the six gates in the AND tree used to form C_{19}^0 plus the AND circuits required to form all other C_{19}^i (i from 1 to 18), terms. Since many of the AND network terms have been formed in C_{19}^0, only two additional gates are required for each i in C_{19}^i; one to create an initial term and one to collect all terms. Actually, we ignore a number of cases where only one additional gate is required. Then the C_{19} AND network consists of $6 + 2 \cdot 18 = 42$ gates. So far, we have ignored fan-out limitations, and it is worth noting that many terms are heavily used—up to 20 times. However, careful design using consolidated terms (gates) where appropriate can keep the fan-out down to about 10—probably a practical maximum. Thus, fan-out limits the use of C_{19} terms in C_{18}, etc. But the size of the AND trees decreases for intermediate bits C_j; e.g., for C_9 about 13 gates are required. As a conservative estimate, assume that (1/2)(42) gates are required as an average for the AND networks. The total number of gates is then:

AND networks:	(1/2)(42)(20)	= 420	
OR networks:		= 80	
initial terms:	2×20	= 40	
Exclusive ORs:	2×20	= 40	
total:		= 580	gates

While 580 gates (closer to 450 with a more detailed count) is high compared to a 20-bit CLA addition, the biggest drawbacks to canonic addition are fan-out and topology, not cost. The high average gate fan-out coupled with relatively congested layout problems leads to an almost three-dimensional communication structure within the AND trees. Both serve to significantly increase average gate delay. Still, canonic addition is an interesting algorithm with practical significance in those cases where at least one operand is limited in size.

3.1.5 Ling Adders

Adders can be developed to recognize the ability of certain circuit families to perform special logic functions very rapidly. The classic case of this is the ability to "DOT" gates together. Here, the output of AND gates (usually) can simply be wired together, giving an OR function. This wired OR or DOT OR has no additional gate delay (although a small additional loading delay is experienced per wired output, due to a line capacitance). Another circuit family feature of interest is complementary outputs: each gate has both the expected (true) output and another complemented output. The widely used current switching (sometimes called emitter coupled or current mode) circuit family incorporates both features. Of course, using the DOT feature may invalidate the premise of the (r, d) circuit model that all logic decisions have unit delay with fan-in r. Ling [16] has carefully developed adder structures to capitalize on the DOT OR ability of these circuits. By encoding pairs of digit positions $(A_i, B_i, A_{i-1}, B_{i-1})$, Ling redefines our notion of sum and carry. To somewhat oversimplify Ling's approach, we attribute the local (lower neighbor) carry enable terms (P_{i-1}) to the definition of the sum (S_i), leaving a reduced synthetic carry (designated H_{i+1}) for non-local carry propagation $(P_i = A_i + B_i)$. Ling finds that the sum (S_i) at bit i can be written as:

$$\begin{aligned} S_i = & (H_{i+1} \oplus P_i) + G_i \cdot H_i \cdot P_{i-1} \\ = & (G_i + P_{i-1} \cdot H_i) \oplus P_i + G_i \cdot H_i \cdot P_{i-1}\text{'} \end{aligned}$$

where H_i is defined by the recursion

$$H_{i+1} = G_i + H_i \cdot P_{i-1}.$$

While the combinatorics of the derivation are formidable, the validity of the above can also be seen from the following table. Note that the recursion is equivalent to:

$$H_{i+1} = G_i + C_i$$

compared with

$$C_{i+1} = G_i + P_i C_i.$$

Now

$$\begin{aligned} P_i \cdot H_{i+1} &= P_i G_i + P_i C_i \\ &= G_i + P_i C_i \\ &= C_{i+1} \end{aligned}$$

or

$$C_i = P_{i-1} \cdot H_i,$$

and

$$H_{i+1} = G_i + C_i = G_i + P_{i-1} H_i.$$

Also, since

$$S_i = A_i \oplus B_i \oplus C_i,$$

then

$$S_i = A_i \oplus B_i \oplus P_{i-1} H_i.$$

Function	Inputs			Outputs	
$f(n)$	A_i	B_i	H_i	S_i	H_{i+1}
0	0	0	0	0	0
1	0	0	1	P_{i-1}	P_{i-1}
2	0	1	0	1	0
3	0	1	1	$\overline{P}_{i-1}$	P_{i-1}
4	1	0	0	1	0
5	1	0	1	$\overline{P}_{i-1}$	P_{i-1}
6	1	1	0	0	1
7	1	1	1	P_{i-1}	1

H_i is conditioned by P_{i-1} in determining the equivalent of C_i. If a term in the table has $H_i = 0$, the equivalent C_i must be zero and the S_i determination can be directly made (as in the cases $f(0), f(2), f(4), f(6)$). Now whenever $H_i = 1$ determines the sum outcome, the P_{i-1} dependency must be introduced. For $f(1)$ and $f(7)$ the $S_i = 1$ if $P_{i-1} = 1$; for $f(3)$ and $f(5)$, the $S_i = 0$ if $P_{i-1} = 1$ (i.e., $f(3)$ and $f(5)$ are conditioned by $\overline{P}_{i-1}$). A direct expansion of the minterms of

$$S_i = (G_i + P_{i-1} \cdot H_i) \oplus P_i + G_i \cdot H_i \cdot P_{i-1}$$

produces the S_i output in the above table:

$$S_i = P_{i-1} \cdot (f(1) + f(7)) + \overline{P}_{i-1} \cdot (f(3) + f(5)) + f(2) + f(4).$$

The synthetic carry H_{i+1} has similar dependency on P_{i-1}; for $f(3)$ and $f(5)$, $H_{i+1} = 1$ occurs if $P_{i-1} = 1$. For $f(6)$ and $f(7)$ $H_{i+1} = 1$ regardless of the $H_i \cdot P_{i-1}$, since $G_i = 1$. The $f(1)$ term is an interesting "don't care" term introduced to simplify the H_{i+1} structure. This $f(4)$ term in H_i does not affect S_i, since S_i depends on H_i. Now S_{i+1} cannot be affected by $H_{i+1}\,(f(1))$, since $P_i\,(f(1)) = 0$. Similarly H_{i+2} also contains the term $H_{i+1}P_i$, which for $f(1)$ is zero by action of P_i.

To understand the advantage of the Ling adder, consider the conventional C_4 (carry-out of bit 3), as contrasted with H_4:

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0,$$

$$H_4 = G_3 + P_2G_2 + P_2P_1G_1 + P_2P_1P_0G_0.$$

Without the DOT function C_4 is implementable ($r = 4$) in three gate delays (two shown, plus one for either P or G). C_4 can be expanded in terms of the input arguments:

$$\begin{aligned} C_4 &= A_3B_3 + (A_3 + B_3)A_2B_2 + (A_3 + B_3)(A_2 + B_2)A_1B_1 \\ &\quad +(A_3 + B_3)(A_2 + B_2)(A_1 + B_1)A_0B_0 \end{aligned}$$

$$\begin{aligned} C_4 &= A_3B_3 + A_3A_2B_2 + B_3A_2B_2 + A_3A_2A_1B_1 \\ &\quad +A_3B_2A_1B_1 + B_3A_2A_1B_1 + B_3B_2A_1B_1 \\ &\quad +A_3A_2A_1A_0B_0 + A_3A_2B_1A_0B_0 + A_3B_2A_1A_0B_0 \\ &\quad +A_3B_2B_1A_0B_0 + B_3A_2A_1A_0B_0 + B_3A_2B_1A_0B_0 \\ &\quad +B_3B_2A_1A_0B_0 + B_3B_2B_1A_0B_0 \end{aligned}$$

If we designate s as the maximum number of lines that can be dotted, then we see that to perform C_4 in one dotted gate delay requires $r = 5$ and $s = 15$.

Now consider the expansion of H_4:

$$\begin{aligned} H_4 &= A_3B_3 + (A_2 + B_2)A_2B_2 + (A_2 + B_2)(A_1 + B_1)A_1B_1 \\ &\quad +(A_2 + B_2)(A_1 + B_1)(A_0 + B_0)A_0B_0 \end{aligned}$$

$$H_4 = A_3B_3 + A_2B_2 + A_2A_1B_1 + B_2A_1B_1$$

$$+\mathrm{A_2A_1A_0B_0} + \mathrm{A_2B_1A_0B_0}$$
$$+\mathrm{B_2A_1A_0B_0} + \mathrm{B_2B_1A_0B_0}$$

Thus, the Ling structure provides one dotted gate delay with $r = 4$ and $s = 8$.

Higher order H look-ahead can be derived in a similar fashion by defining a fan-in limited I term as the conjunction of P_is; e.g.,

$$I_7 = P_6P_5P_4P_3.$$

Rather than dot ORing the summands to form the P_i term, the bipolar nature of the ECL circuit can be used to form the OR in one gate delay:

$$P_i = A_i + B_i$$

and the P_i terms can be dot ANDed to form the I terms. Thus, I_7, I_{11}, and I_{15} can be found with one gate delay.

Suppose we designate the pseudo-carryout of each four bit group as H'_{16}, H'_{12}, H'_8, H'_4, and the group carry-generate as G_4, G_8, G_{12}. Then

$$\begin{aligned}
H_4 &= H'_4 \\
H_8 &= H'_8 + I_7H'_4 \\
H_{12} &= H'_{12} + I_{11}H'_8 + I_{11}I_7H'_4 \\
H_{16} &= H'_{16} + I_{15}H'_{12} + I_{15}I_{11}H'_8 + I_{15}I_{11}I_7H'_4.
\end{aligned}$$

Of course,

$$\begin{aligned}
C_{16} &= P_{15}H_{16} \\
&= P_{15}(H'_{16} + I_{15}H'_{12} + I_{15}I_{11}H'_8 + I_{15}I_{11}I_7H'_4),
\end{aligned}$$

since terms such as

$$I_7H'_4 = P_6P_5P_4P_3H_4 = P_6P_5P_4C_4.$$

Thus,

$$I_{15}I_{11}I_7H'_4 = \prod_{i=4}^{14} P_i \cdot C_4, \qquad G'_4 = C_4.$$

Similarly,

$$\begin{aligned}
I_{15}I_{11}H'_8 &= I_{15}P_{10}P_9P_8P_7H'_8 \\
&= I_{15}P_{10}P_9P_8G'_8 \\
&= \prod_{i=8}^{14} P_i \cdot G'_8.
\end{aligned}$$

Ling suggests that the conditional sum algorithm be used in forming the final result. Thus, S_{31} through S_{16} is found for both $C_{16} = 0$ and $C_{16} = 1$; these results are gated with the appropriate

	Fixed Radix (Binary)				
	Winograd's lower bound	Conditional sum	Carry-look-ahead	Canonic	Ling
Formula	$\lceil \log_r 2n \rceil$	$5 + 2\left\lceil \log_{r-1}\left(\lceil \frac{n}{r} \rceil - 1\right)\right\rceil$	$4\lceil \log_r n \rceil$	$2\lceil \log_r n - 1 \rceil + 2 - \delta$	$\lceil \log_r \frac{n}{2} \rceil + 1$
gate delays $n = 64$ bits $r =$ fan-in $= 5$	4	9	12	8	4*

	Variable Radix (Residue)	
	Winograd's lower bound	ROM look-up table
Formula	$\lceil \log_r 2\lceil \log_d \alpha(n) \rceil \rceil$	$2 + \lceil \log_r m \rceil + \lceil \log_r 2^m \rceil$
gate delays n $= 64$ bits $r =$ fan-in $= 5$	$d = 2$, $\alpha(> 2^n) = 59$,	$m = \lceil \log_d \alpha(> 2^n) \rceil = 6$
	2	7

* The Ling adder requires dot OR of 16 terms and assumes no additional delay for such dotting.

Table 3.1: Comparison of addition speed (in gate delay) of the various hardware realizations and the lower bounds of Winograd.

true value of C_{16} and dot ORed in one gate delay. The "extra" delay forming C_{16} from $P_{15}H_{16}$ adds no delay, since P_{15} is ANDed with the sum selector MUX function as below:

$$\begin{aligned} S &= S_E C_{16} + S_N \overline{C_{16}} \\ S &= S_E P_{15} H_{16} + S_N(\overline{P_{15}} + \overline{H_{16}}) \\ &= S_E P_{15} H_{16} + S_N \overline{P_{15}} + S_N \overline{H_{16}}, \end{aligned}$$

where S_E and S_N represent the higher order 16-bit sum with and without carry-in. (ECL circuits have complementary outputs; $\overline{H_{16}}$ is always available.) Thus, the Ling adder can realize a sum delay in:

$$\boxed{\text{Ling gate delays} = \lceil \log_r \tfrac{n}{2} \rceil + 1} \quad ,$$

so long as the gates can be dotted with capability $2^{r-1} \leq s$.

3.1.6 Simultaneous Addition of Multiple Operands: Carry-Save Adders.

Frequently, more than two operands (positive or negative) are to be summed in the minimum time. In fact, this is the basic requirement of multiplication. Clearly, one can do better than simply summing a pair and then adding each additional operand to the previous sum. Consider the following decimal example:

Carry-	176	
Saving	324	
Addition	$\underline{558}$	
Carry-	948	Column sum
Propagating	$\underline{11\ }$	Column carry
Addition	1058	Total

Regardless of the number of entries to be summed, summation can proceed simultaneously on all columns generating a pair of numbers: column sum and column carry. These numbers must be

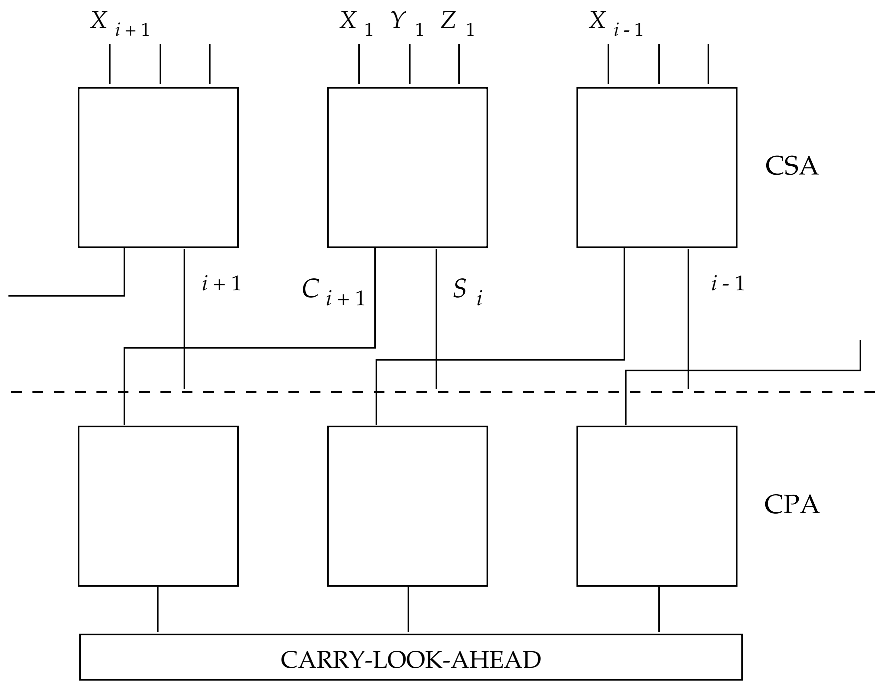


Figure 3.7: Addition of three n-bit numbers.

added with carry propagation. Thus, it should be possible to reduce the addition of any number of operands to a *carry-propagating addition* of only two: Column sum and Column carry. Of course, the generation of these two column operands may take some time, but this should be significantly less than the serial operand by operand propagating addition.

Consider the addition of three n-bit binary numbers. We refer to the structure that sums a column as a *carry-save adder* (CSA). That is, the CSA will take 3 bits of the same significance and produce the sum (same significance) and the carry (1 bit higher significance). Note that this is exactly what a 1-bit position of a *binary full adder* does; but the input connections are different between CSA and binary full adder. Suppose we wish to add X, Y, and Z. Let X_i, Y_i, and Z_i represent the ith-bit position.

We thus have the desired structure: the binary-full adder. However, instead of chaining the carry-out signal from a lower order position to the carry-in input, the third operand is introduced to the "carry-in" and the output produces two operands which now must be summed by a propagating adder. Binary full adders when used in this way are called carry-save adders (CSA). Thus, to add three numbers we require only two additional gate delays (the CSA delay) in excess of the carry propagate adder delay.

The same technique can be extended to more than three operand addition by cascading CSAs.

Suppose we wish to add W, X, Y, Z; the ith-bit position might be implemented as in Figure 3.8. High-speed multiplication depends on rapid addition of multiples of the multiplicand and, as we shall see in the next chapter, uses a generalization of the carry-save adder technique.

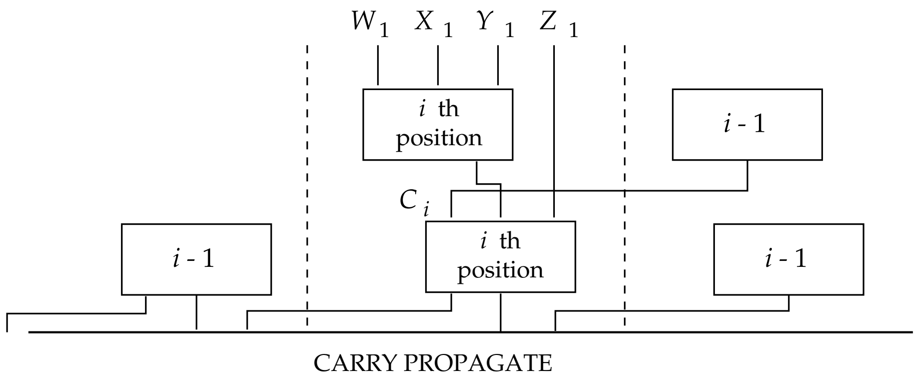


Figure 3.8: Addition of four n-bit numbers.

3.2 Exercises

1. What is the gate delay of a 24-bit adder for the following implementations ($r = 4$)?
   (a) CLA.
   (b) Conditional sum.
   (c) Canonic adder.

2. (a) Suppose $r = 4$ and the maximum dot-OR capability is also 4; for a 64-bit addition the Ling adder will require how many unit delays, while a canonic adder requires how many unit delays?
   (b) If six 32-bit operands are to be added simultaneously, how many unit delays are required in CSAs before two operands can be added in a CPA?
   (c) In a certain machine, the execution time for floating point addition is greater than that for floating point multiplication. Explain (i.e., state the delay conditions which lead to this situation).

3. The System 370 effective address computation involves the addition of three unsigned numbers, two of 24 bits and one of 12 low order bits.
   (a) Design a fast adder for this address computation (an overflow is an invalid address).
   (b) Extend your adder to accommodate a fast comparison of the effective address with a 24^{bit} upper bound address.

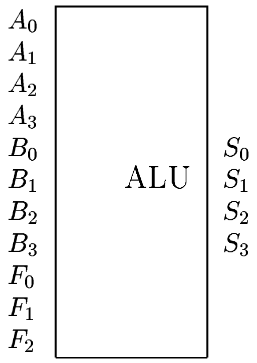
4. Design a circuit that can be connected to a 4-bit ALU to detect 2's complement arithmetic overflow. The ALU takes two 4-bit input operands and provides a 4-bit output result defined by three function bits. The ALU can perform eight functions, defined by $F_2F_1F_0$. The object is to make the circuit as fast as possible. Use as many gates as you like.

   For gate timing, use the following:

NAND, NOR, NOT:	5 units
OR, AND:	7 units
XOR:	10 units

Note: Assume delay through ALU $\gg$ single gate delay.

F_2	F_1	F_0	Function
0	0	0	0
0	0	1	$B - A$
0	1	0	$A - B$
0	1	1	$A + B$
1	0	0	$A \oplus B$
1	0	1	A OR B
1	1	0	A AND B
1	1	1	-1



5. It has been suggested that $1^K \times 8^b$ ROMs could be used as counters to realize a 32-bit CPA ($C_{\text{in}} = 0$). Design such a unit. Show each counter and its DOT configuration or $(C_R, \ldots, d)$ designation. Clearly show how its input and output relate to other counters.

   Minimize (with priority):

   (a) The number of ROM delays.

   (b) The total number of ROMs.

   (c) The number of ROM types.

   What are the number of ROM delays, the total number of ROMs, and the number of ROM types? Show a complete design.