6 Carry-Lookahead Adders

"Computers can figure out all kinds of problems, except the things in the world that just don't add up."
ANONYMOUS

Adder designs considered in Chapter 5 have worst-case delays that grow at least linearly with the word width k. Since the most-significant bit of the sum is a function of all the $2k$ input bits, assuming that the gate fan-in is limited to d, a lower bound on addition latency is $\log_d(2k)$. An interesting question, therefore, is whether one can add two k-bit binary numbers in $O(\log k)$ worst-case time. Carry-lookahead adders, covered in this chapter, represent a commonly used scheme for logarithmic time addition. Other schemes are introduced in Chapter 7.

6.1 UNROLLING THE CARRY RECURRENCE

Recall the g_i (generate), p_i (propagate), a_i (annihilate or absorb), and t_i (transfer) auxiliary signals introduced in Section 5.6:

$g_i = 1$ iff $x_i + y_i \geq r$	Carry is generated
$p_i = 1$ iff $x_i + y_i = r - 1$	Carry is propagated
$t_i = \overline{a}_i = g_i \vee p_i$	Carry is not annihilated

These signals, along with the carry recurrence

$$c_{i+1} = g_i \vee p_i c_i = g_i \vee t_i c_i$$

allow us to decouple the problem of designing a fast carry network from details of the number system (radix, digit set). In fact it does not even matter whether we are adding or subtracting; any carry network can be used as a borrow network if we simply redefine the preceding signals to correspond to borrow generation, borrow propagation, and so on.

The carry recurrence $c_{i+1} = g_i \vee p_i c_i$ states that a carry will enter stage $i + 1$ if it is generated in stage i or it enters stage i and is propagated by that stage. One can easily unroll this recurrence, eventually obtaining each carry c_i as a logical function of the operand bits and c_{in}. Here are three steps of the unrolling process for c_i:

$$\begin{aligned}
c_i &= g_{i-1} \vee c_{i-1}p_{i-1} \\
&= g_{i-1} \vee (g_{i-2} \vee c_{i-2}p_{i-2})p_{i-1} = g_{i-1} \vee g_{i-2}p_{i-1} \vee c_{i-2}p_{i-2}p_{i-1} \\
&= g_{i-1} \vee g_{i-2}p_{i-1} \vee g_{i-3}p_{i-2}p_{i-1} \vee c_{i-3}p_{i-3}p_{i-2}p_{i-1} \\
&= g_{i-1} \vee g_{i-2}p_{i-1} \vee g_{i-3}p_{i-2}p_{i-1} \vee g_{i-4}p_{i-3}p_{i-2}p_{i-1} \vee c_{i-4}p_{i-4}p_{i-3}p_{i-2}p_{i-1}
\end{aligned}$$

The unrolling can be continued until the last product term contains $c_0 = c_{\text{in}}$. The unrolled version of the carry recurrence has the following simple interpretation: carry enters into position i if and only if a carry is generated in position $i - 1$ (g_{i-1}), or a carry generated in position $i - 2$ is propagated by position $i - 1$ $(g_{i-2}p_{i-1})$, or a carry generated in position $i - 3$ is propagated at $i - 2$ and $i - 1$ $(g_{i-3}p_{i-2}p_{i-1})$, etc.

After full unrolling, we can compute all the carries in a k-bit adder directly from the auxiliary signals (g_i, p_i) and c_{in}, using two-level AND-OR logic circuits with maximum gate fan-in of $k + 1$. For $k = 4$, the logic expressions are as follows:

$$\begin{aligned}
c_4 &= g_3 \vee g_2p_3 \vee g_1p_2p_3 \vee g_0p_1p_2p_3 \vee c_0p_0p_1p_2p_3 \\
c_3 &= g_2 \vee g_1p_2 \vee g_0p_1p_2 \vee c_0p_0p_1p_2 \\
c_2 &= g_1 \vee g_0p_1 \vee c_0p_0p_1 \\
c_1 &= g_0 \vee c_0p_0
\end{aligned}$$

Here, c_0 and c_4 are the 4-bit adder's c_{in} and c_{out}, respectively. A carry network based on the preceding equations can be used in conjunction with two-input ANDs, producing the g_i signals, and two-input XORs, producing the p_i and sum bits, to build a 4-bit binary adder. Such an adder is said to have *full carry lookahead.*

Note that since c_4 does not affect the computation of the sum bits, it can be derived based on the simpler equation

$$c_4 = g_3 \vee c_3p_3$$

with little or no speed penalty. The resulting carry network is depicted in Fig. 6.1.

Clearly, full carry lookahead is impractical for wide words. The fully unrolled carry equation for c_{31}, for example, consists of 32 product terms, the largest of which contains

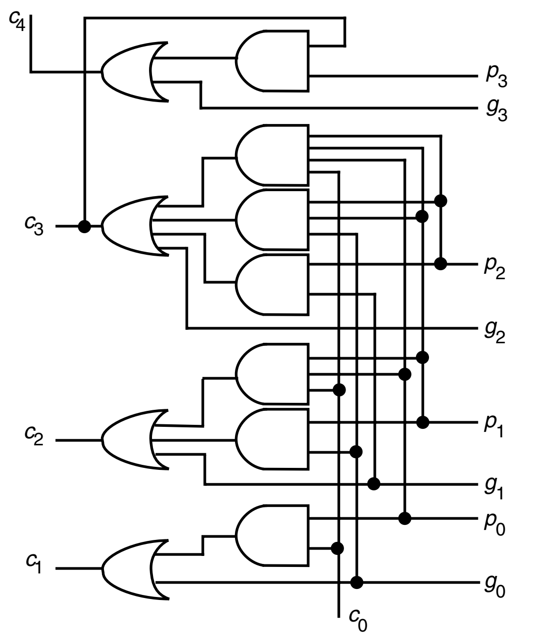


Figure 6.1 A 4-bit carry network with full lookahead.

32 literals. Thus, the required AND and OR functions must be realized by tree networks, leading to increased latency and cost. Two schemes for managing this complexity immediately suggest themselves:

High-radix addition (i.e., radix 2^h)
Multilevel lookahead

High-radix addition increases the latency for generating the auxiliary signals and sum digits but simplifies the carry network. Depending on the implementation method and technology, an optimal radix might exist. Multilevel lookahead is the technique used in practice and is covered in Section 6.2.

6.2 CARRY-LOOKAHEAD ADDER DESIGN

Consider radix-16 addition of two binary numbers that are characterized by their g_i and p_i signals. For each radix-16 digit position, extending from bit position i to bit position $i + 3$ of the original binary numbers (where i is a multiple of 4), "block generate" and "block propagate" signals can be derived as follows:

$$g_{[i,i+3]} = g_{i+3} \vee g_{i+2}p_{i+3} \vee g_{i+1}p_{i+2}p_{i+3} \vee g_i p_{i+1}p_{i+2}p_{i+3}$$

$$p_{[i,i+3]} = p_i p_{i+1} p_{i+2} p_{i+3}$$

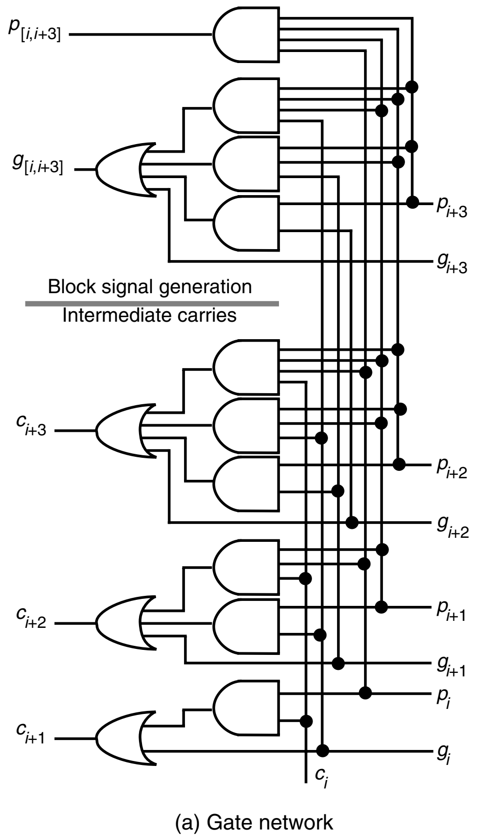
The preceding equations can be interpreted in the same way as unrolled carry equations: the four bit positions collectively propagate an incoming carry c_i if and only if each of the four positions propagates; they collectively generate a carry if a carry is produced in position $i+3$, or it is produced in position $i+2$ and propagated by position $i+3$, etc.

If we replace the c_4 portion of the carry network of Fig. 6.1 with circuits that produce the block generate and propagate signals $g_{[i,i+3]}$ and $p_{[i,i+3]}$, the 4-bit *lookahead carry generator* of Fig. 6.2a is obtained. Figure 6.2b shows the 4-bit lookahead carry generator in schematic form. We will see shortly that such a block can be used in a multilevel structure to build a carry network of any desired width.

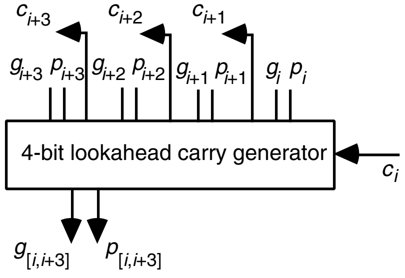
First, however, let us take a somewhat more general view of the block generate and propagate signals. Assuming $i_0 < i_1 < i_2$, we can write

$$g_{[i_0,i_2-1]} = g_{[i_1,i_2-1]} \vee g_{[i_0,i_1-1]}p_{[i_1,i_2-1]}$$

This equation essentially says that a carry is generated by the block of positions from i_0 to $i_2 - 1$ if and only if a carry is generated by the $[i_1, i_2 - 1]$ block or a carry generated



(a) Gate network



c_{i+3}
c_{i+2}
c_{i+1}
g_{i+3} p_{i+3} g_{i+2} p_{i+2} g_{i+1} p_{i+1} g_i p_i
4-bit lookahead carry generator
c_i
$g_{[i,i+3]}$
$p_{[i,i+3]}$

(b) Schematic diagram

Figure 6.2 A 4-bit lookahead carry generator.

by the $[i_0, i_1 - 1]$ block is propagated by the $[i_1, i_2 - 1]$ block. Similarly

$$p_{[i_0,i_2-1]} = p_{[i_0,i_1-1]}p_{[i_1,i_2-1]}$$

In fact the two blocks being merged into a larger block do not have to be contiguous; they can also be overlapping. In other words, for the possibly overlapping blocks $[i_1,j_1]$ and $[i_0,j_0]$, $i_0 \leq i_1 - 1 \leq j_0 < j_1$, we have

$$g_{[i_0,j_1]} = g_{[i_1,j_1]} \vee g_{[i_0,j_0]}p_{[i_1,j_1]}$$

$$p_{[i_0,j_1]} = p_{[i_0,j_0]}p_{[i_1,j_1]}$$

Figure 6.3 shows that a 4-bit lookahead carry generator can be used to combine the g and p signals from adjacent or overlapping blocks into the p and g signals for the combined block.

Given the 4-bit lookahead carry generator of Fig. 6.2, it is an easy matter to synthesize wider adders based on a multilevel carry-lookahead scheme. For example, to construct a two-level 16-bit carry-lookahead adder, we need four 4-bit adders and a 4-bit lookahead carry generator, connected together as shown on the upper right quadrant of Fig. 6.4. The 4-bit lookahead carry generator in this case can be viewed as predicting the three intermediate carries in a 4-digit radix-16 addition. The latency through this 16-bit adder consists of the time required for:

Producing the g and p signals for individual bit positions (1 gate level).

Producing the g and p signals for 4-bit blocks (2 gate levels).

Predicting the carry-in signals c_4, c_8, and c_{12} for the blocks (2 gate levels).

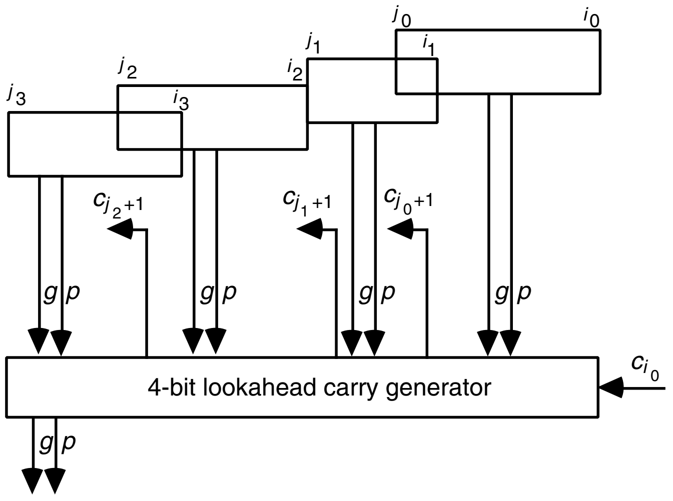


Figure 6.3 Combining of g and p signals of four (contiguous or overlapping) blocks of arbitrary widths into the g and p signals for the overall block $[i_0, j_3]$.

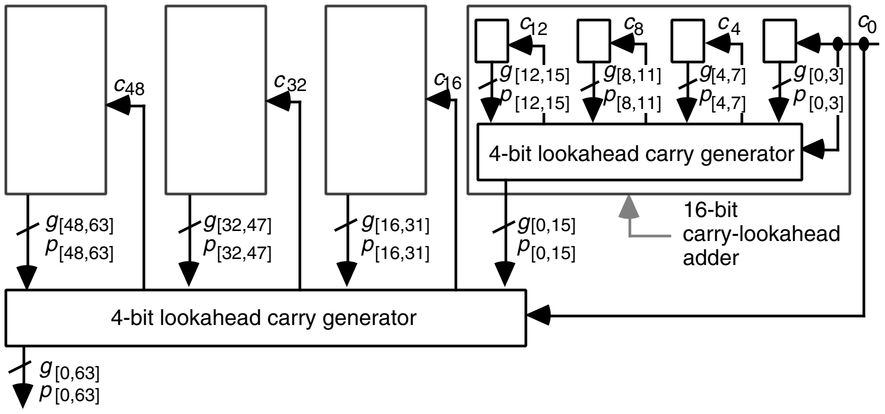


Figure 6.4 Building a 64-bit carry-lookahead adder from 16 4-bit adders and 5 lookahead carry generators.

Predicting the internal carries within each 4-bit block (2 gate levels).

Computing the sum bits (2 gate levels).

Thus the total latency for the 16-bit adder is 9 gate levels, which is much better than the 32 gate levels required by a 16-bit ripple-carry adder.

Similarly, to construct a three-level 64-bit carry-lookahead adder, we can use four of the 16-bit adders above plus one 4-bit lookahead carry generator, connected together as shown in Fig. 6.4. The delay will increase by four gate levels with each additional level of lookahead: two levels in the downward movement of the g and p signals, and two levels for the upward propagation of carries through the extra level. Thus, the delay of a k-bit carry-lookahead adder based on 4-bit lookahead blocks is

$$T_{\text{lookahead}-\text{add}} = 4 \log_4 k + 1 \text{ gate levels}$$

Hence, the 64-bit carry-lookahead adder of Fig. 6.4 has a latency of 13 gate levels.

One can of course use 6-bit or 8-bit lookahead blocks to reduce the number of lookahead levels for a given word width. But this may not be worthwhile in view of the longer delays introduced by gates with higher fan-in. When the word width is not a power of 4, some of the inputs and/or outputs of the lookahead carry generators remain unused, and the latency formula becomes $4\lceil\log_4 k\rceil + 1$.

One final point about the design depicted in Fig. 6.4: this 64-bit adder does not produce a carry-out signal (c_{64}), which would be needed in many applications. There are two ways to remedy this problem in carry-lookahead adders. One is to generate c_{out} externally based on auxiliary signals or the operand and sum bits in position $k-1$:

$$c_{\text{out}} = g_{[0,k-1]} \vee c_0 p_{[0,k-1]} = x_{k-1}y_{k-1} \vee \bar{s}_{k-1}(x_{k-1} \vee y_{k-1})$$

Another is to design the adder to be 1 bit wider than needed (e.g., 61 bits instead of 60), using the additional sum bit as c_{out}.

6.3 LING ADDER AND RELATED DESIGNS

The Ling adder is a type of carry-lookahead adder that achieves significant hardware savings. Consider the carry recurrence and its unrolling by four steps:

$$\begin{aligned} c_i &= g_{i-1} \vee c_{i-1}p_{i-1} = g_{i-1} \vee c_{i-1}t_{i-1} \\ &= g_{i-1} \vee g_{i-2}t_{i-1} \vee g_{i-3}t_{i-2}t_{i-1} \vee g_{i-4}t_{i-3}t_{i-2}t_{i-1} \vee c_{i-4}t_{i-4}t_{i-3}t_{i-2}t_{i-1} \end{aligned}$$

Ling's modification consists of propagating $h_i = c_i \vee c_{i-1}$ instead of c_i. To understand the following derivations, we note that g_{i-1} implies c_i ($c_i = 1$ if $g_{i-1} = 1$), which in turn implies h_i.

$$\begin{aligned} c_{i-1}p_{i-1} &= c_{i-1}p_{i-1} \vee g_{i-1}p_{i-1} \text{ \{zero\} } \vee p_{i-1}c_{i-1}p_{i-1} \text{ \{repeated term\}} \\ &= c_{i-1}p_{i-1} \vee (g_{i-1} \vee p_{i-1}c_{i-1})p_{i-1} \\ &= (c_{i-1} \vee c_i)p_{i-1} = h_i p_{i-1} \\ c_i &= g_{i-1} \vee c_{i-1}p_{i-1} \\ &= h_i g_{i-1} \text{\{since } g_{i-1} \text{ implies } h_i\text{\}} \vee h_i p_{i-1} \text{ \{from above\}} \\ &= h_i(g_{i-1} \vee p_{i-1}) = h_i\, t_{i-1} \\ h_i &= c_i \vee c_{i-1} = (g_{i-1} \vee c_{i-1}p_{i-1}) \vee c_{i-1} \\ &= g_{i-1} \vee c_{i-1} = g_{i-1} \vee h_{i-1}t_{i-2} \text{ \{from above\}} \end{aligned}$$

Unrolling the preceding recurrence for h_i, we get

$$\begin{aligned} h_i &= g_{i-1} \vee t_{i-2}\, h_{i-1} = g_{i-1} \vee t_{i-2}(g_{i-2} \vee h_{i-2}\, t_{i-3}) \\ &= g_{i-1} \vee g_{i-2} \vee h_{i-2}\, t_{i-2}\, t_{i-3} \text{ \{since } t_{i-2}\, g_{i-2} = g_{i-2}\text{\}} \\ &= g_{i-1} \vee g_{i-2} \vee g_{i-3}\, t_{i-3}\, t_{i-2} \vee h_{i-3}\, t_{i-4}\, t_{i-3}\, t_{i-2} \\ &= g_{i-1} \vee g_{i-2} \vee g_{i-3}\, t_{i-2} \vee g_{i-4}\, t_{i-3}\, t_{i-2} \vee h_{i-4}\, t_{i-4}\, t_{i-3}\, t_{i-2} \end{aligned}$$

We see that expressing h_i in terms of h_{i-4} needs five product terms, with a maximum four-input AND gate, and a total of 14 gate inputs. By contrast, expressing c_i as

$$c_i = g_{i-1} \vee g_{i-2}t_{i-1} \vee g_{i-3}t_{i-2}t_{i-1} \vee g_{i-4}t_{i-3}t_{i-2}t_{i-1} \vee c_{i-4}t_{i-4}t_{i-3}t_{i-2}t_{i-1}$$

requires five terms, with a maximum five-input AND gate, and a total of 19 gate inputs. The advantage of h_i over c_i is even greater if we can use wired-OR (3 gates with 9 inputs vs. 4 gates with 14 inputs). Once h_i is known, however, the sum is obtained by a slightly more complex expression compared with $s_i = p_i \oplus c_i$:

$$s_i = p_i \oplus c_i = p_i \oplus h_i t_{i-1}$$

This concludes our presentation of Ling's improved carry-lookahead adder. The reader can skip the rest of this section with no harm to continuity.

A number of related designs have been developed based on ideas similar to Ling's. For example, Doran [Dora88] suggests that one can in general propagate η instead of c where

$$\eta_{i+1} = f(x_i, y_i, c_i) = \psi(x_i, y_i)c_i \vee \phi(x_i, y_i)\bar{c}_i$$

The residual functions ψ and ϕ in the preceding Shannon expansion of f around c_i must be symmetric, and there are but eight symmetric functions of the two variables x_i and y_i. Doran shows that not all $8 \times 8 = 64$ possibilities are valid choices for ψ and ϕ, since in some cases the sum cannot be computed based on the η_i values. Dividing the eight symmetric functions of x_i and y_i into the two disjoint subsets $\{0, \bar{t}_i, g_i, \bar{p}_i\}$ and $\{1, t_i, \bar{g}_i, p_i\}$, Doran proves that ψ and ϕ cannot both belong to the same subset. Thus, there are only 32 possible adders. Four of these 32 possible adders have the desirable properties of Ling's adder, which represents the special case of $\psi(x_i, y_i) = 1$ and $\phi(x_i, y_i) = g_i = x_i y_i$.

6.4 CARRY DETERMINATION AS PREFIX COMPUTATION

Consider two contiguous or overlapping blocks B′ and B″ and their associated generate and propagate signal pairs (g', p') and (g'', p''), respectively. As shown in Fig. 6.5, the generate and propagate signals for the merged block B can be obtained from the equations:

$$g = g'' \vee g'p''$$
$$p = p'p''$$

That is, carry generation in the larger group takes place if the left group generates a carry or the right group generates a carry and the left one propagates it, while propagation occurs if both groups propagate the carry.

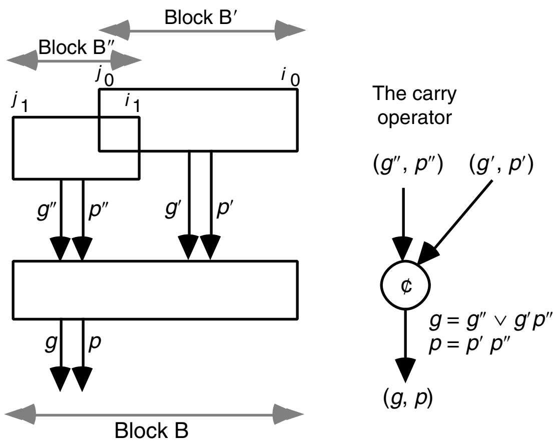


Figure 6.5 Combining of g and p signals of two (contiguous or overlapping) blocks B′ and B″ of arbitrary widths into the g and p signals for the overall block B.

We note that in the discussion above, the indices i_0, j_0, i_1, and j_1 defining the two contiguous or overlapping blocks are in fact immaterial, and the same expressions can be written for any two adjacent groups of any width. Let us define the "carry" operator $¢$ on (g, p) signal pairs as follows (right side of Fig. 6.5):

$$(g,p) = (g',p')\, ¢\, (g'',p'') \text{ means } g = g'' \vee g'p'', \;\; p = p'p''$$

The carry operator $¢$ is *associative*, meaning that the order of evaluation does not affect the value of the expression $(g',p')\, ¢\, (g'',p'')\, ¢\, (g''',p''')$, but it is not *commutative*, since $g'' \vee g'p''$ is in general not equal to $g' \vee g''p'$.

Observe that in an adder with no c_{in}, we have $c_{i+1} = g_{[0,i]}$; that is, a carry enters position $i+1$ if and only if one is generated by the block $[0, i]$. In an adder with c_{in}, a carry-in of 1 can be viewed as a carry generated by stage -1; we thus set $p_{-1} = 0, g_{-1} = c_{\text{in}}$ and compute $g_{[-1,i]}$ for all i. So, the problem remains the same, but with an extra stage ($k + 1$ rather than k). The problem of carry determination can, therefore, be formulated as follows:

Given				
(g_0, p_0)	(g_1, p_1)	$\cdots$	(g_{k-2}, p_{k-2})	(g_{k-1}, p_{k-1})
Find				
$(g_{[0,0]}, p_{[0,0]})$	$(g_{[0,1]}, p_{[0,1]})$	$\cdots$	$(g_{[0,k-2]}, p_{[0,k-2]})$	$(g_{[0,k-1]}, p_{[0,k-1]})$

The desired signal pairs can be obtained by evaluating all the prefixes of

$$(g_0, p_0)\, ¢\, (g_1, p_1)\, ¢ \cdots ¢\, (g_{k-2}, p_{k-2})\, ¢\, (g_{k-1}, p_{k-1})$$

in parallel. In this way, the carry problem is converted to a parallel prefix computation, and any prefix computation scheme can be used to find all the carries.

A parallel prefix computation can be defined with any associative operator. In the following, we use the addition operator with integer operands, in view of its simplicity and familiarity, to illustrate the methods. The *parallel prefix sums* problem is defined as follows:

Given:	x_0	x_1	x_2	x_3	$\cdots$	x_{k-1}
Find:	x_0	$x_0 + x_1$	$x_0 + x_1 + x_2$	$x_0 + x_1 + x_2 + x_3$	$\cdots$	$x_0 + x_1 + \cdots + x_{k-1}$

Any design for this parallel prefix sums problem can be converted to a carry computation network by simply replacing each adder cell with the carry operator of Fig. 6.5. There is one difference worth mentioning, though. Addition is commutative. So if prefix sums are obtained by computing and combining the partial sums in an arbitrary manner, the resulting design may be unsuitable for a carry network. However, as long as blocks whose sums we combine are always contiguous and we do not change their ordering, no problem arises.

Just as one can group numbers in any way to add them, (g, p) signal pairs can be grouped in any way for combining them into block signals. In fact, (g, p) signals give us an additional flexibility in that overlapping groups can be combined without affecting the outcome, whereas in addition, use of overlapping groups would lead to incorrect sums.

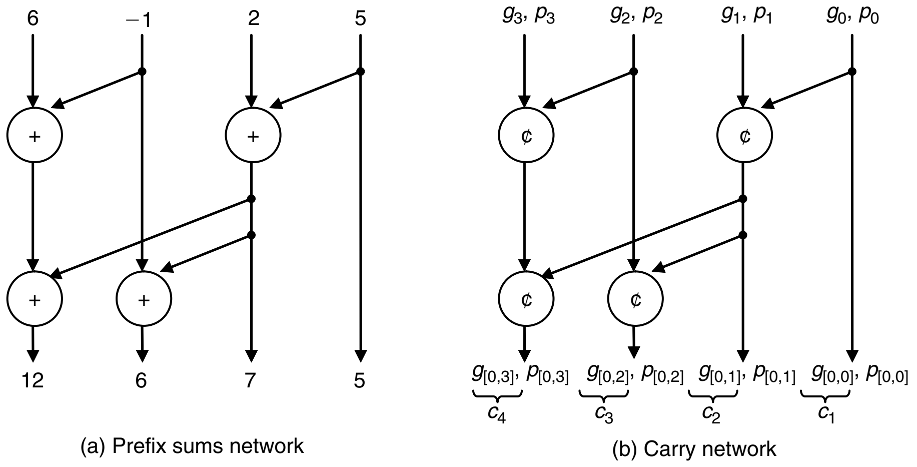


Figure 6.6 Four-input parallel prefix sums network and its corresponding carry network.

Figure 6.6a depicts a four-input prefix sums network composed of four adder blocks, arranged in two levels. It produces the prefix sums 5, 7, 6, and 12 when supplied with the inputs 5, 2, −1, and 6, going from right to left. Note that we use the right-to-left ordering of inputs and outputs on diagrams, because this corresponds to how we index digit positions in positional number representation. So, what we are computing really constitutes postfix sums of the expression $x_3 + x_2 + x_1 + x_0$. However, we will continue to use the terms "prefix sums" and "parallel prefix networks" in accordance with the common usage. As long as we remember that the indexing in carry network diagrams goes from right to left, no misinterpretation will arise. Figure 6.6b shows the carry network derived from the prefix sums network of Fig. 6.6a by replacing each adder with a carry operator. It also shows how the outputs of this carry network are related to carries that we need to complete a 4-bit addition.

6.5 ALTERNATIVE PARALLEL PREFIX NETWORKS

Now, focusing on the problem of computing prefix sums, we can use several strategies to synthesize a parallel prefix sum network. Figure 6.7 is based on a divide-and-conquer approach as proposed by Ladner and Fischer [Ladn80]. The low-order $k/2$ inputs are processed by the subnetwork at the right to compute the prefix sums $s_0, s_1, \ldots, s_{k/2-1}$. Partial prefix sums are computed for the high-order $k/2$ values (the left subnetwork) and $s_{k/2-1}$ (the leftmost output of the first subnetwork) is added to them to complete the computation. Such a network is characterized by the following recurrences for its delay (in terms of adder levels) and cost (number of adder cells):

Delay recurrence:	$D(k) = D(k/2) + 1 = \log_2 k$
Cost recurrence:	$C(k) = 2C(k/2) + k/2 = (k/2)\log_2 k$

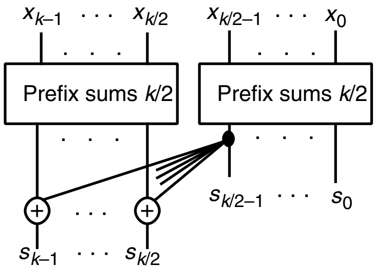


Figure 6.7 Ladner–Fischer parallel prefix sums network built of two $k/2$-input networks and $k/2$ adders.

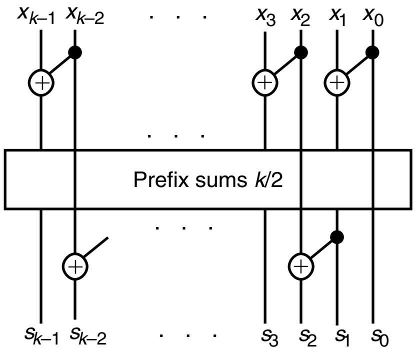

x_{k-1} x_{k-2} . . . x_3 x_2 x_1 x_0
Prefix sums $k/2$
s_{k-1} s_{k-2} . . . s_3 s_2 s_1 s_0

Figure 6.8 Parallel prefix sums network built of one $k/2$-input network and $k - 1$ adders.

A second divide-and-conquer design for computing prefix sums, proposed by Brent and Kung [Bren82], is depicted in Fig. 6.8. Here, the inputs are first combined pairwise to obtain the following sequence of length $k/2$:

$$x_0 + x_1 \qquad x_2 + x_3 \qquad x_4 + x_5 \qquad \cdots \qquad x_{k-4} + x_{k-3} \qquad x_{k-2} + x_{k-1}$$

Parallel prefix sum computation on this new sequence yields the odd-indexed prefix sums $s_1, s_3, s_5, \ldots$ for the original sequence. Even-indexed prefix sums are then computed by using $s_{2j} = s_{2j-1} + x_{2j}$. The cost and delay recurrences for the design of Fig. 6.8 are:

Delay recurrence:	$D(k) = D(k/2) + 2 = 2 \log_2 k - 1$
	actually we will see later that $D(k) = 2 \log_2 k - 2$
Cost recurrence:	$C(k) = C(k/2) + k - 1 = 2k - 2 - \log_2 k$

So, the Ladner–Fischer design is faster than the Brent–Kung design ($\log_2 k$ as opposed to $2 \log_2 k - 2$ adder levels) but also much more expensive [$(k/2) \log_2 k$ as opposed to $2k - 2 - \log_2 k$ adder cells]. The Ladner–Fischer design also leads to large fan-out requirements if implemented directly in hardware. In other words, the output of one of the adders in the right part must feed the inputs of $k/2$ adders in the left part.

The 16-input instance of the Brent–Kung design of Fig. 6.8 is depicted in Fig. 6.9. Note that even though the graph of Fig. 6.9 appears to have seven levels, two of the levels near the middle are independent, thus implying a single level of delay. In general,

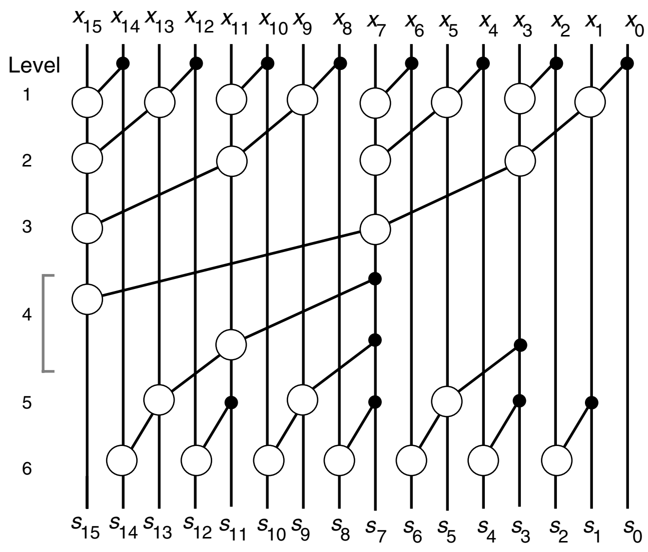

x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
Level
1
2
3
4
5
6
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Figure 6.9
Brent–Kung parallel prefix graph for 16 inputs.

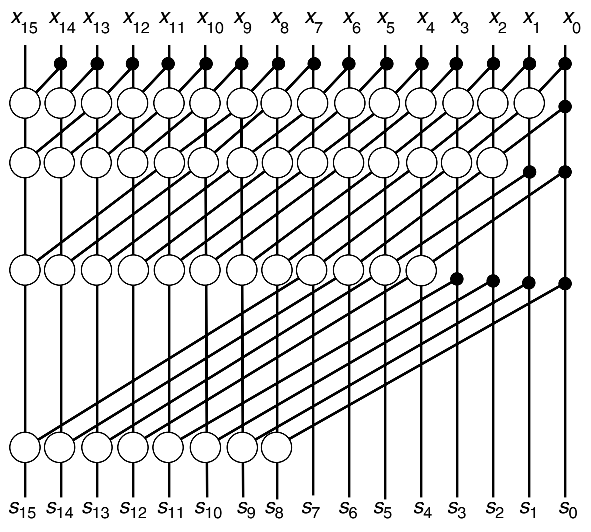

x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Figure 6.10
Kogge–Stone parallel prefix graph for 16 inputs.

a k-input Brent–Kung parallel prefix graph will have a delay of $2\log_2 k - 2$ levels and a cost of $2k - 2 - \log_2 k$ cells.

Figure 6.10 depicts a Kogge–Stone parallel prefix graph that has the same delay as the design shown in Fig. 6.7 but avoids its fan-out problem by distributing the computations. A k-input Kogge–Stone parallel prefix graph has a delay of $\log_2 k$ levels and a cost of $k\log_2 k - k + 1$ cells. The Kogge–Stone parallel prefix graph represents the fastest

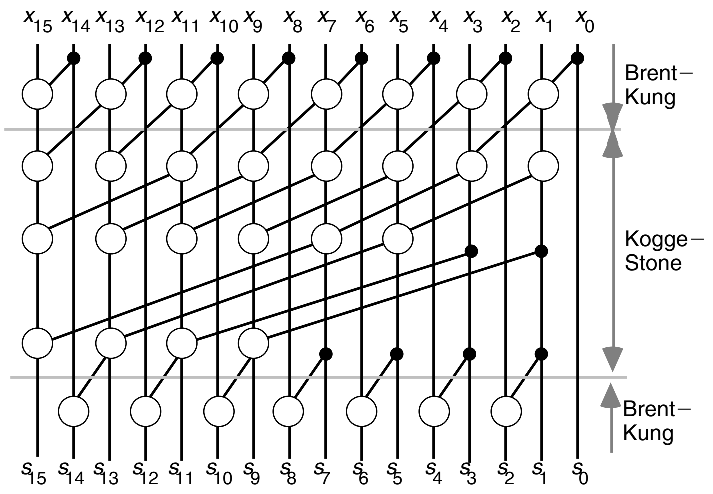

x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
Brent–Kung
Kogge–Stone
Brent–Kung
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Figure 6.11 A hybrid Brent–Kung/Kogge–Stone parallel prefix graph for 16 inputs.

possible implementation of a parallel prefix computation if only two-input blocks are allowed. However, its cost can be prohibitive for large k, in terms of both the number of cells and the dense wiring between them.

Many other parallel prefix network designs are possible. For example, it has been suggested that the Brent–Kung and Kogge–Stone approaches be combined to form hybrid designs [Sugl90]. In Fig. 6.11, the middle four of the six levels in the design of Fig. 6.9 (representing an eight-input parallel prefix computation) have been replaced by the eight-input Kogge–Stone network. The resulting design has five levels and 32 cells, placing it between the pure Brent–Kung (six levels, 26 cells) and pure Kogge–Stone (four levels, 49 cells) designs.

More generally, if a single Brent–Kung level is used along with a $k/2$-input Kogge–Stone design, delay and cost of the hybrid network become $\log_2 k + 1$ and $(k/2)\log_2 k$, respectively. The resulting design is thus close to minimum in terms of delay (only one level more than Kogge–Stone) but costs roughly half as much.

The theory of parallel prefix graphs is quite rich and well developed. There exist both theoretical bounds and actual designs with different restrictions on fan-in/fan-out and with various optimality criteria in terms of cost and delay (see, e.g., Chapters 5–7, pp. 133–211, of [Laks94]).

In devising their design, Brent and Kung [Bren82] were motivated by the need to reduce the chip area in very large-scale integration (VLSI) layout of the carry network. Other performance or hardware limitations may also be considered. The nice thing about formulating the problem of carry determination as a parallel prefix computation is that theoretical results and a wealth of design strategies carry over with virtually no effort. Not all such relationships between carry networks and parallel prefix networks, or the virtually unlimited hybrid combinations, have been explored in full.

6.6 VLSI IMPLEMENTATION ASPECTS

The carry network of Fig. 6.9 is quite suitable for VLSI implementation, but it might be deemed too slow for high-performance designs and/or wide words. Many designers have proposed alternate networks that offer reduced latency by using features of particular technologies and taking advantage of related optimizations. We review one example here that is based on radix-256 addition of 56-bit numbers as implemented in the Advanced Micro Devices Am29050 microprocessor. The following description is based on a 64-bit version of the adder.

In radix-256 addition of 64-bit numbers, only the carries $c_8, c_{16}, c_{24}, c_{32}, c_{40}, c_{48}$, and c_{56} need to be computed. First, 4-bit Manchester carry chains (MCCs) of the type shown in Fig. 6.12a are used to derive g and p signals for 4-bit blocks. These signals, denoted by [0, 3], [4, 7], [8, 11], etc. on the left side of Fig. 6.13, then form the inputs to one 5-bit and three 4-bit MCCs that in turn feed two more MCCs in the third level. The six MCCs in levels 2 and 3 in Fig. 6.13 are of the type shown in Fig. 6.12b; that is, they also produce intermediate g and p signals. For example, the MCC with inputs [16, 19], [20, 23], [24, 27], and [28, 31] yields the intermediate outputs [16, 23] and [16, 27], in addition to the signal pair [16, 31] for the entire group.

Various parallel-prefix adders, all with minimum-latency designs when only node delays are considered, may turn out quite different when the effects of interconnects (including fan-in, fan-out, and signal propagation delay on wires) are considered [Beau01], [Huan00], [Know99].

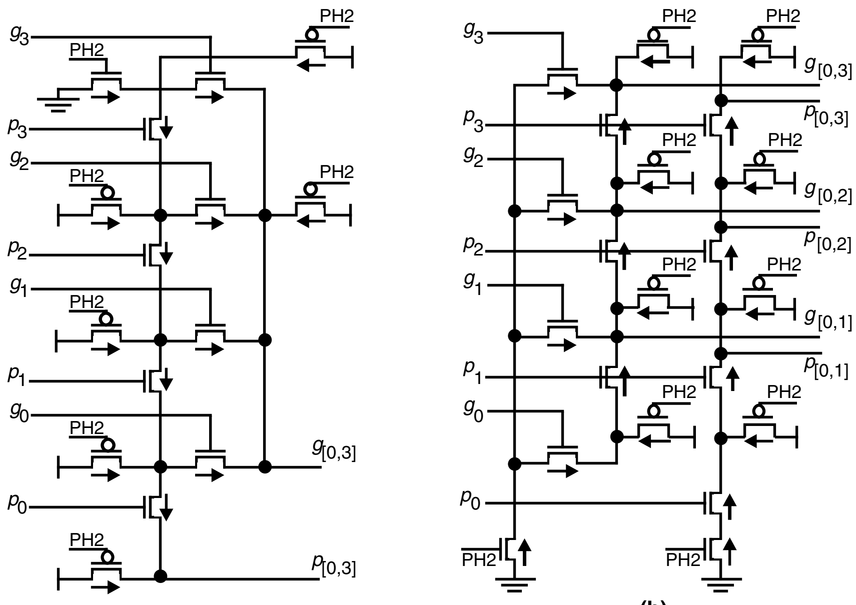


Figure 6.12 Example 4-bit MCC designs in CMOS technology [Lync92].

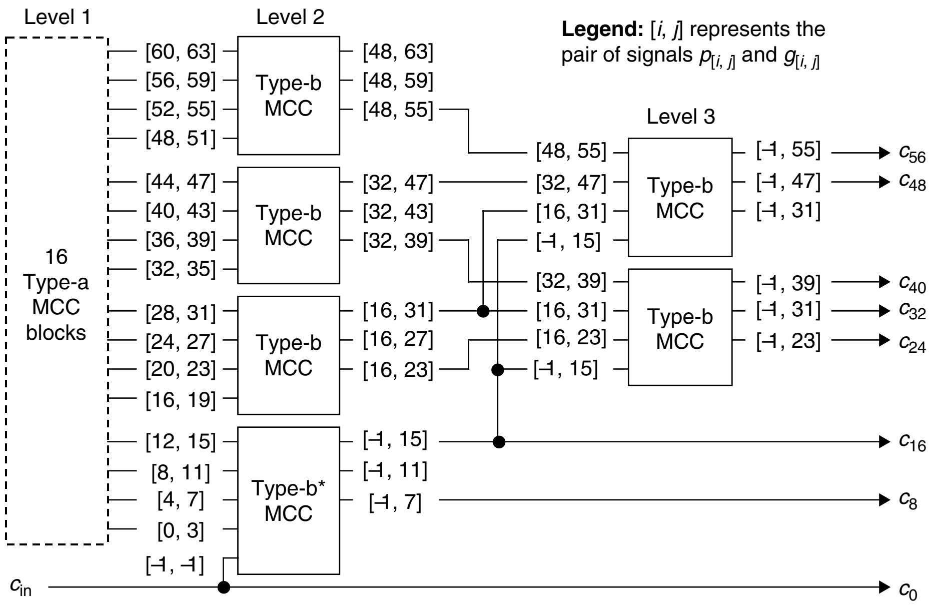


Figure 6.13 Spanning-tree carry-lookahead network. Type-a and Type-b MCCs refer to the circuits of Figs. 6.12a and 6.12b, respectively.

PROBLEMS

6.1 Borrow-lookahead subtractor

We know that any carry network producing the carries c_i based on g_i and p_i signals can be used, with no modification, as a borrow-propagation circuit to find the borrows b_i.

a. Define the borrow-generate γ_i and borrow-propagate π_i signals in general and for the special case of binary operands.

b. Present the design of a circuit to compute the difference digit d_i from γ_i, π_i, and the incoming borrow b_i.

6.2 1's-complement carry-lookahead adder

Discuss how the requirement for end-around carry in 1's-complement addition affects the design and performance of a carry-lookahead adder.

6.3 High-radix carry-lookahead adder

Consider radix-2^h addition of binary numbers and assume that the total time needed for producing the digit g and p signals, and determining the sum digits after all carries are known, equals δh, where δ is a constant. Carries are determined by a multilevel lookahead network using unit-time 2-bit lookahead carry generators. Derive the optimal radix that minimizes the addition latency as a function of δ and discuss.

6.4 Unconventional carry-lookahead adder

Consider the following method for synthesizing a k-bit adder from four $k/4$-bit adders and a 4-bit lookahead carry generator. The $k/4$-bit adders have no group g or p output. Both the g_i and p_i inputs of the lookahead carry generator are connected to the carry-out of the ith $k/4$-bit adder, $0 \le i \le 3$. Intermediate carries of the lookahead carry generator and c_{in} are connected to the carry-in inputs of the $k/4$-bit adders. Will the suggested circuit add correctly? Find the adder's latency or justify your negative answer.

6.5 Decimal carry-lookahead adder

Consider the design of a 15-digit decimal adder for unsigned numbers (width = 60 bits).

a. Design the required circuits for carry-generate and carry-propagate assuming binary-coded decimal digits.
b. Repeat part a with excess-3 encoding for the decimal digits, where digit value a is represented by the binary encoding of $a + 3$.
c. Complete the design of the decimal adder of part b by proposing a carry-lookahead circuit and the sum computation circuit.

6.6 Carry lookahead with overlapped blocks

a. Write down the indices for the g and p signals on Fig. 6.3. Then present expressions for these signals in terms of g and p signals of nonoverlapping subblocks such as $[i_0,\ i_1 - 1]$ and $[i_1, j_0]$.
b. Prove that the combining equations for the g and p signals for two contiguous blocks also apply to overlapping blocks (see Fig. 6.5).

6.7 Latency of a carry-lookahead adder

Complete Fig. 6.4 by drawing boxes for the g and p logic and the sum computation logic. Then draw a critical path on the resulting diagram and indicate the number of gate levels of delay on each segment of the path.

6.8 Ling adder or subtractor

a. Show the complete design of a counterpart to the lookahead carry generator of Fig. 6.2 using Ling's method.
b. How does the design of a Ling subtractor differ from that of a Ling adder? Present complete designs for all the parts that are different.

6.9 Ling-type adders

Based on the discussion at the end of Section 6.3, derive one of the other three Ling-type adders proposed by Doran [Dora88]. Compare the derived adder with a Ling adder.

6.10 Fixed-priority arbiters

A fixed-priority arbiter has k request inputs $R_{k-1}, \ldots, R_1, R_0$, and k grant outputs G_i. At each arbitration cycle, at most one of the grant signals is 1 and that corresponds to the highest-priority request signal (i.e., $G_i = 1$ if and only if $R_i = 1$ and $R_j = 0$ for $j < i$).

a. Design a synchronous arbiter using ripple-carry techniques. *Hint:* Consider $c_0 = 1$ along with carry propagation and annihilation rules; there is no carry generation.

b. Design the arbiter using carry-lookahead techniques. Determine the number of lookahead levels required with 64 inputs and estimate the total arbitration delay.

6.11 Carry-lookahead incrementer

a. Design a 16-bit incrementer using the carry-lookahead principle.

b. Repeat part a using Ling's approach.

c. Compare the designs of parts a and b with respect to delay and cost.

6.12 Parallel prefix networks

Find delay and cost formulas for the Brent–Kung and Kogge–Stone designs when the word width k is not a power of 2.

6.13 Parallel prefix networks

a. Draw Brent–Kung, Kogge–Stone, and hybrid parallel prefix graphs for 12, 20, and 24 inputs.

b. Using the results of part a, plot the cost, delay, and cost-delay product for the five types of networks for $k = 12, 16, 20, 24, 32$ bits and discuss.

6.14 Hybrid carry-lookahead adders

a. Find the depth and cost of a 64-bit hybrid carry network with two levels of the Brent–Kung scheme at each end and the rest built by the Kogge–Stone construction.

b. Compare the design of part a to pure Brent–Kung and Kogge–Stone schemes and discuss.

6.15 Parallel prefix networks

a. Obtain delay and cost formulas for a hybrid parallel prefix network that has l levels of Brent–Kung design at the top and bottom and a $k/2^l$-input Kogge–Stone network in the middle.

b. Use the delay-cost-product figure of merit to find the best combination of the two approaches for word widths from 8 to 64 (powers of 2 only).

6.16 Speed and cost limits for carry computation

Consider the computation of c_i, the carry into the ith stage of an adder, based on the g_j and t_j signals using only two-input AND and OR gates. Note that only the computation of c_i, independent of other carries, is being considered.

a. What is the minimum possible number of AND/OR gates required?
b. What is the minimum possible number of gate levels in the circuit?
c. Can one achieve the minima of parts a and b simultaneously? Explain.

6.17 Variable-block carry-lookahead adders

Study the benefits of using nonuniform widths for the MCC blocks in a carry-lookahead adder of the type discussed in Section 6.6 [Kant93].

6.18 Implementing the carry operator

Show that the carry operator of Fig. 6.5 can be implemented by using $g = (g' \vee g'')(p'' \vee g'')$, thereby making all signals for p and g go through two levels of logic using a NOT-NOR or NOR-NOR implementation.

6.19 Parallel prefix networks

a. Formulate the carry-computation problem as an instance of the parallel prefix problem.
b. Using as few two-input adder blocks as possible, construct a prefix sums network for 8 inputs. Label the inputs x_0, x_1, x_2, etc., and the outputs $s_{[0,0]}, s_{[0,1]}, s_{[0,2]}$, etc.
c. Show the design of the logic block that should replace the adders in part b if your prefix sums network is to be converted to an 8-bit-wide carry-lookahead network.
d. What do you need to add to your carry network so that it accommodates a carry-in signal?

6.20 Parallel prefix networks

In the divide-and-conquer scheme of Fig. 6.7 for designing a parallel prefix network, one may observe that all but one of the outputs of the right block can be produced one time unit later without affecting the overall latency of the network. Show that this observation leads to a linear-cost circuit for k-input parallel prefix computation with $\lceil \log_2 k \rceil$ latency. *Hint:* Define type-x prefix circuits, $x \geq 0$, that produce their leftmost output with $\lceil \log_2 k \rceil$ latency and all other outputs with latencies not exceeding $\lceil \log_2 k \rceil + x$, where k is the number of inputs. Write recurrences that relate $C_x(k)$ for such circuits [Ladn80].

6.21 Carry-lookahead adders

Consider an 8-bit carry-lookahead adder with 2-bit blocks. Assume that block p and g signals are produced after three gate delays and that each block uses ripple-carry internally. The design uses a 4-bit lookahead carry generator with two gate delays.

Carry ripples through each stage in two gate delays and sum bits are computed in two gate delays once all the internal carries are known. State your assumptions whenever the information provided is not sufficient to answer the question.

a. Compute the total addition time, in terms of gate delays, for this 8-bit adder.
b. We gradually increase the adder width to 9, 10, 11, . . . bits using four ripple-carry groups of equal or approximately equal widths, while keeping the block p and g delay constant. At what word width k would it be possible to increase the adder speed by using an additional level of lookahead?

6.22 Asynchronous carry computation

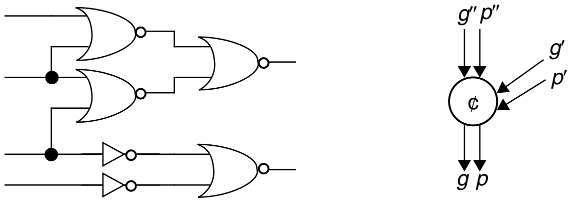
Show that by combining the best-case O(1) delay of an asynchronous ripple-carry adder with the worst-case O(log k) delay of a lookahead design, and using whichever result arrives first, an O(log log k) average-time asynchronous adder can be built [Mano98].

6.23 Carry-lookahead adders

Design a 64-bit carry-lookahead adder that yields both the sum of its inputs and the sum plus *ulp*. Such an adder is useful as part of a floating-point adder, because it allows rounding to be performed with no further carry propagation [Burg99]. *Hint:* Parallel-prefix carry networks already produce the information that would be needed to add *ulp* to the sum without any carry propagation.

6.24 Implementing the carry operator

Show that the logic circuit on the left below implements the carry operator, shown on the right, and label the inputs and outputs of the circuit accordingly. What advantage do you see for this circuit compared with the AND-OR version implied by Fig. 6.5?



6.25 Designing fast comparators

Given a fast carry network of any design, show how it can be used to build a fast comparator to determine whether $x > y$, where x and y are unsigned integers.

6.26 Alternative formulation of carry-lookahead addition

Our discussion of carry determination as prefix computation was based on (g, p) signal pairs. A similar development can be based on (g, a) signal pairs.

a. Redo Section 6.4 of the book, including Fig. 6.5, using the alternative formulation above.
b. Show that a ripple-carry type parallel prefix circuit that uses (g, a) pairs leads to an adder design similar to that in Fig. 5.9.
c. Discuss the advantages and drawbacks of this alternative formulation.

6.27 Parallel prefix Ling adders

Consider the unrolled Ling recurrence $h_4 = g_4 \vee g_3 \vee t_3g_2 \vee t_3t_2g_1 \vee t_3t_2t_1g_0$. Show that the latter formula is equivalent to $h_4 = (g_4 \vee g_3) \vee (t_3t_2)(g_2 \vee g_1) \vee (t_3t_2)(t_1t_0)g_0$. Similarly, we have $h_5 = (g_5 \vee g_4) \vee (t_4t_3) \vee (g_3 \vee g_2) \vee (t_4t_3)(t_2t_1)(g_1 \vee g_0)$. Discuss how these new formulations lead to parallel prefix Ling adders in which odd and even "Ling carries" are computed separately and with greater efficiency [Dimi05].

REFERENCES AND FURTHER READINGS

[Bayo83] Bayoumi, M. A., G. A. Jullien, and W. C. Miller, "An Area-Time Efficient NMOS Adder," *Integration: The VLSI Journal*, Vol. 1, pp. 317–334, 1983.

[Beau01] Beaumont-Smith, A., and C.-C. Lim, "Parallel Prefix Adder Design," *Proc. 15th Symp. Computer Arithmetic*, pp. 218–225, 2001.

[Bren82] Brent, R. P., and H. T. Kung, "A Regular Layout for Parallel Adders," *IEEE Trans. Computers*, Vol. 31, pp. 260–264, 1982.

[Burg99] Burgess, N., and S. Knowles, "Efficient Implementation of Rounding Units", *Proc. 33rd Asilomar Conf. Signals Systems and Computers*, pp. 1489–1493, 1999.

[Burg05] Burgess, N., "New Models of Prefix Adder Topologies," *J. VLSI Signal Processing*, Vol. 40, pp. 125–141, 2005.

[Dimi05] Dimitrakopoulos, G., and D. Nikolos, "High-Speed Parallel-Prefix VLSI Ling Adders," *IEEE Trans. Computers*, Vol. 54, No. 2, pp. 225–231, 2005.

[Dora88] Doran, R. W., "Variants of an Improved Carry Look-Ahead Adder," *IEEE Trans. Computers*, Vol. 37, No. 9, pp. 1110–1113, 1988.

[Han87] Han, T., and D. A. Carlson, "Fast Area-Efficient Adders," *Proc. 8th Symp. Computer Arithmetic*, pp. 49–56, 1987.

[Harr03] Harris, D., "A Taxonomy of Parallel Prefix Networks," *Proc. 37th Asilomar Conf. Signals, Systems, and Computers*, Vol. 2, pp. 2213–2217, 2003.

[Huan00] Huang, Z., and M. D. Ercegovac, "Effect of Wire Delay on the Design of Prefix Adders in Deep-Submicron Technology," *Proc. 34th Asilomar Conf. Signals, Systems, and Computers*, October 2000, pp. 1713–1717, 2000.

[Kant93] Kantabutra, V., "A Recursive Carry-Lookahead/Carry-Select Hybrid Adder," *IEEE Trans. Computers*, Vol. 42, No. 12, pp. 1495–1499, 1993.

[Know99] Knowles, S., "A Family of Adders," *Proc. 14th Symp. Computer Arithmetic*, 1999, printed at the end of ARITH-15 Proceedings, pp. 277–284, 2001.

[Kogg73] Kogge, P. M. and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrences," *IEEE Trans. Computers,* Vol. 22, pp. 786–793, 1973.

[Ladn80] Ladner, R. E., and M. J. Fischer, “Parallel Prefix Computation,” *J. ACM*, Vol. 27, No. 4, pp. 831–838, 1980.

[Laks94] Lakshmivarahan, S., and S. K. Dhall, *Parallel Computing Using the Prefix Problem*, Oxford University Press, 1994.

[Ling81] Ling, H., “High-Speed Binary Adder,” *IBM J. Research and Development*, Vol. 25, No. 3, pp. 156–166, 1981.

[Lync92] Lynch, T., and E. Swartzlander, “A Spanning Tree Carry Lookahead Adder,” *IEEE Trans. Computers*, Vol. 41, No. 8, pp. 931–939, 1992.

[Mano98] Manohar, R., and J. A. Tierno, "Asynchronous Parallel Prefix Computation," *IEEE Trans. Computers*, Vol. 47, No. 11, pp. 1244–1252, 1998.

[Ngai84] Ngai, T. F., M. J. Irwin, and S. Rawat, “Regular Area-Time Efficient Carry-Lookahead Adders,” *J. Parallel and Distributed Computing*, Vol. 3, No. 3, pp. 92–105, 1984.

[Sugl90] Sugla, B., and D. A. Carlson, “Extreme Area-Time Tradeoffs in VLSI,” *IEEE Trans. Computers*, Vol. 39, No. 2, pp. 251–257, 1990.

[Wei90] Wei, B. W. Y., and C. D. Thompson, “Area-Time Optimal Adder Design,” *IEEE Trans. Computers*, Vol. 39, No. 5, pp. 666–675, 1990.

[Wein56] Weinberger, A., and J. L. Smith, “A One-Microsecond Adder Using One-Megacycle Circuitry,” *IRE Trans. Computers*, Vol. 5, pp. 65–73, 1956.