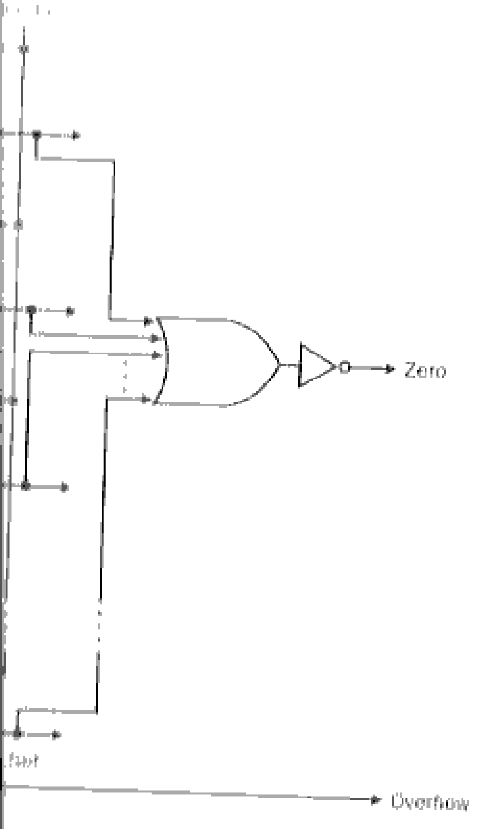


4 Zero detector to Figure 4.18

ontrol lines Bnegate and Operation and the

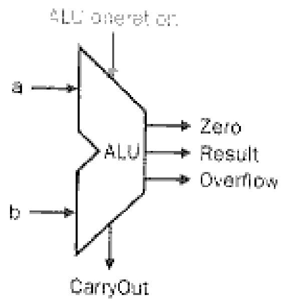


FIGURE 4.21 The symbol commonly used to represent an ALU, as shown in Figure 4.19. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

Carry Lookahead

The next question is, How quickly can this ALU add two 32-bit operands? We can determine the a and b inputs, but the CarryIn input depends on the operation in the adjacent 1-bit adder. If we trace all the way through the chain of dependencies, we connect the most significant bit to the least significant bit, so the most significant bit of the sum must wait for the *sequential* evaluation of all 32 1-bit adders. This sequential chain reaction is too slow to be used in time-critical hardware.

There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the $\log_2$ of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry.

A key to understanding fast carry schemes is to remember that, unlike software, hardware executes in parallel whenever inputs change.

Fast Carry Using "Infinite" Hardware

Appendix B mentions that any equation can be represented in two levels of logic. Since the only external inputs are the two operands and the CarryIn to the least significant bit of the adder, in theory we could calculate the CarryIn values to all the remaining bits of the adder in just two levels of logic.

For example, the CarryIn for bit 2 of the adder is exactly the CarryOut of bit 1, so the formula is

$$\text{CarryIn2} = (\text{b1} \cdot \text{CarryIn1}) + (\text{a1} \cdot \text{CarryIn1}) + (\text{a1} \cdot \text{b1})$$

Similarly, CarryIn1 is defined as

$$\text{CarryIn1} = (\text{b0} \cdot \text{CarryIn0}) + (\text{a0} \cdot \text{CarryIn0}) + (\text{a0} \cdot \text{b0})$$

Using the shorter and more traditional abbreviation of *ci* for CarryIn*i*, we can rewrite the formulas as

$$c2 = (b1 \cdot c1) + (a1 \cdot c1) + (a1 \cdot b1)$$
$$c1 = (b0 \cdot c0) + (a0 \cdot c0) + (a0 \cdot b0)$$

Substituting the definition of c1 for the first equation results in this formula:

$$\begin{aligned} c2 = {} & (a1 \cdot a0 \cdot b0) + (a1 \cdot a0 \cdot c0) + (a1 \cdot b0 \cdot c0) \\ & + (b1 \cdot a0 \cdot b0) + (b1 \cdot a0 \cdot c0) + (b1 \cdot b0 \cdot c0) + (a1 \cdot b1) \end{aligned}$$

You can imagine how the equation expands as we get to higher bits in the adder; it grows exponentially with the number of bits. This complexity is reflected in the cost of the hardware for fast carry, making this simple scheme prohibitively expensive for wide adders.

Fast Carry Using the First Level of Abstraction: Propagate and Generate

Most fast carry schemes limit the complexity of the equations to simplify the hardware, while still making substantial speed improvements over ripple carry. One such scheme is a *carry-lookahead adder*. In Chapter 1, we said computer systems cope with complexity by using levels of abstraction. A carry-lookahead adder relies on levels of abstraction in its implementation.

Let's factor our original equation as a first step:

$$\begin{aligned} ci{+}1 &= (bi \cdot ci) + (ai \cdot ci) + (ai \cdot bi) \\ &= (ai \cdot bi) + (ai + bi) \cdot ci \end{aligned}$$

If we were to rewrite the equation for c2 using this formula, we would see some repeated patterns:

$$c2 = (a1 \cdot b1) + (a1 + b1) \cdot ((a0 \cdot b0) + (a0 + b0) \cdot c0)$$

Note the repeated appearance of $(ai \cdot bi)$ and $(ai + bi)$ in the formula above. These two important factors are traditionally called *generate* (g*i*) and *propagate* (p*i*):

$$gi = ai \cdot bi$$
$$pi = ai + bi$$

Using them to define *ci+1*, we get

$$ci{+}1 = gi + pi \cdot ci$$

To see where the signals get their names, suppose g*i* is 1. Then

$$ci{+}1 = gi + pi \cdot ci = 1 + pi \cdot ci = 1$$

That is, the adder *generates* a CarryOut ($ci+1$) independent of the value of CarryIn (ci). Now suppose that gi is 0 and pi is 1. Then

$$ci+1 = gi + pi \cdot ci = 0 + 1 \cdot ci = ci$$

That is, the adder *propagates* CarryIn to a CarryOut. Putting the two together, CarryIn$i+1$ is a 1 if either gi is 1 or both pi is 1 and CarryIni is 1.

As an analogy, imagine a row of dominoes set on edge. The end domino can be tipped over by pushing one far away provided there are no gaps between the two. Similarly, a carry out can be made true by a generate far away provided all the propagates between them are true.

Relying on the definitions of propagate and generate as our first level of abstraction, we can express the CarryIn signals more economically. Let's show it for 4 bits:

$$\begin{aligned}
c1 &= g0 + (p0 \cdot c0) \\
c2 &= g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0) \\
c3 &= g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0) \\
c4 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\
&\quad + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)
\end{aligned}$$

These equations just represent common sense: CarryIni is a 1 if some earlier adder generates a carry and all intermediary adders propagate a carry. Figure 4.22 uses plumbing to try to explain carry lookahead.

Even this simplified form leads to large equations and, hence, considerable logic even for a 16-bit adder. Let's try moving to two levels of abstraction.

Fast Carry Using the Second Level of Abstraction

First we consider this 4-bit adder with its carry-lookahead logic as a single building block. If we connect them in ripple carry fashion to form a 16-bit adder, the add will be faster than the original with a little more hardware.

To go faster, we'll need carry lookahead at a higher level. To perform carry lookahead for 4-bit adders, we need propagate and generate signals at this higher level. Here they are for the four 4-bit adder blocks:

$$\begin{aligned}
P0 &= p3 \cdot p2 \cdot p1 \cdot p0 \\
P1 &= p7 \cdot p6 \cdot p5 \cdot p4 \\
P2 &= p11 \cdot p10 \cdot p9 \cdot p8 \\
P3 &= p15 \cdot p14 \cdot p13 \cdot p12
\end{aligned}$$

That is, the "super" propagate signal for the 4-bit abstraction (Pi) is true only if each of the bits in the group will propagate a carry.

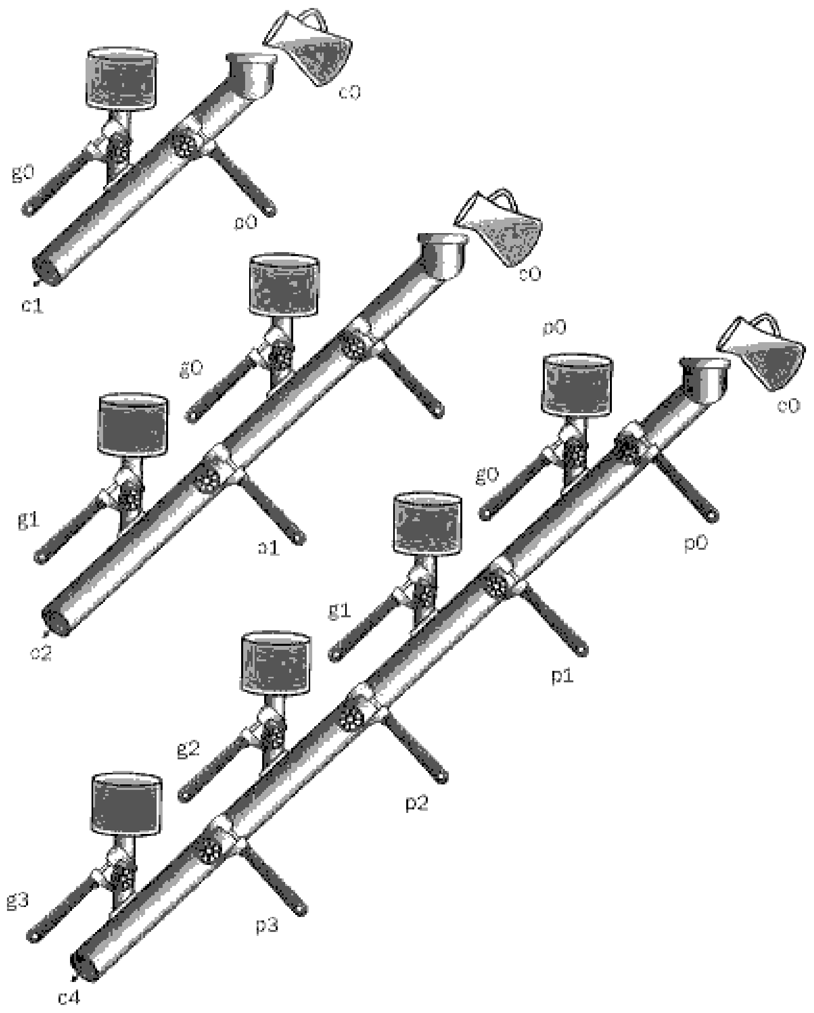


FIGURE 4.22 A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water, pipes, and valves. The wrenches are turned to open and close valves. Water is shown in color. The output of the pipe (*ci+1*) will be full if either the nearest generate value (*gi*) is turned on or if the *i* propagate value (*pi*) is on and there is water further upstream, either from an earlier generate, or propagate with water behind it. CarryIn (c0) can result in a carry out without the help of any generates, but with the help of *all* propagates.

For the "super" generate signal (*Gi*), we care only if there is a carry out of the most significant bit of the 4-bit group. This obviously occurs if generate is true for that most significant bit; it also occurs if an earlier generate is true *and* all the intermediate propagates, including that of the most significant bit, are also true:

$G0 = g3 + (p3 \cdot g2) + (p$

$G1 = g7 + (p7 \cdot g6) + (p$

$G2 = g11 + (p11 \cdot g10) +$

$G3 = g15 + (p15 \cdot g14) +$

Figure 4.23 updates our plu

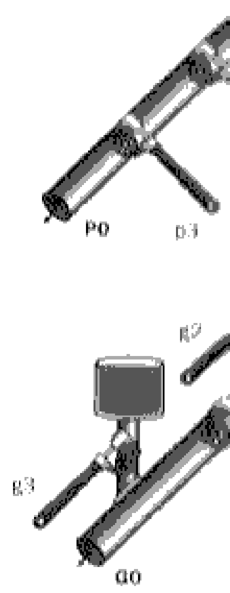
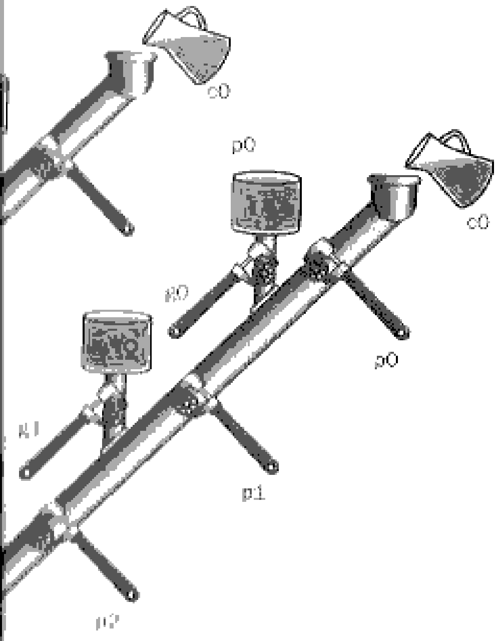


FIGURE 4.23 A plumbing analogy
P0 is open only if all four propagate
generate (*gi*) is open and all the prop



carry lookahead for 1 bit, 2 bits, and 4 bits using
are turned to open and close valves. Water is shown to
full if either the nearest generate value (gi) is turned on
there is water further upstream, either from an earlier
i or. Carry-in (c0) can result in a carry out without the
all propagates.

(Gi), we care only if there is a carry out of
group. This obviously occurs if generate is
also occurs if an earlier generate is true and
cluding that of the most significant bit, are

$$
\begin{aligned}
G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)\\
G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)\\
G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)\\
G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)
\end{aligned}
$$

Figure 4.23 updates our plumbing analogy to show P0 and G0.

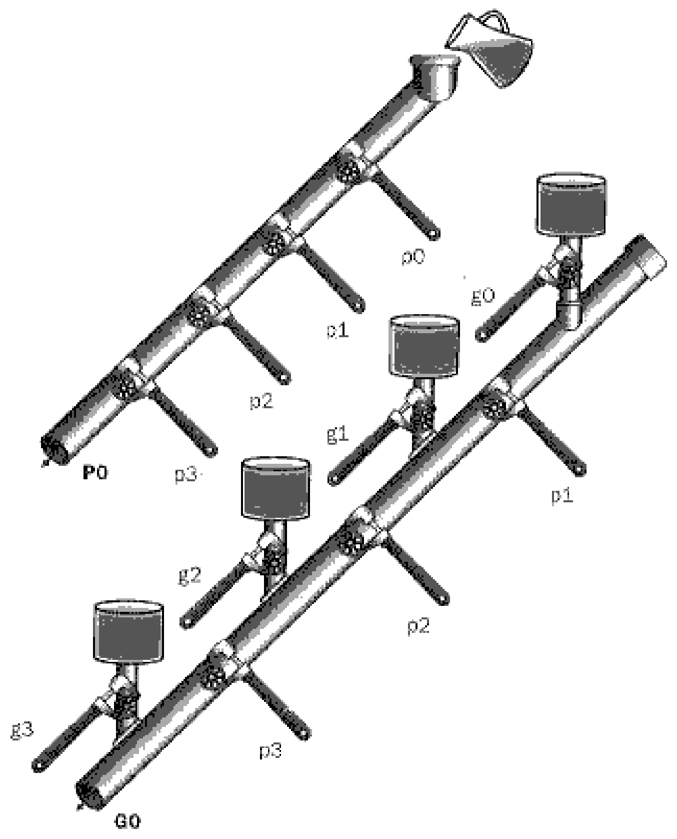


FIGURE 4.23 A plumbing analogy for the next-level carry-lookahead signals P0 and G0. P0 is open only if all four propagates (pi) are open, while water flows in G0 only if at least one generate (gi) is open and all the propagates downstream from that generate are open.

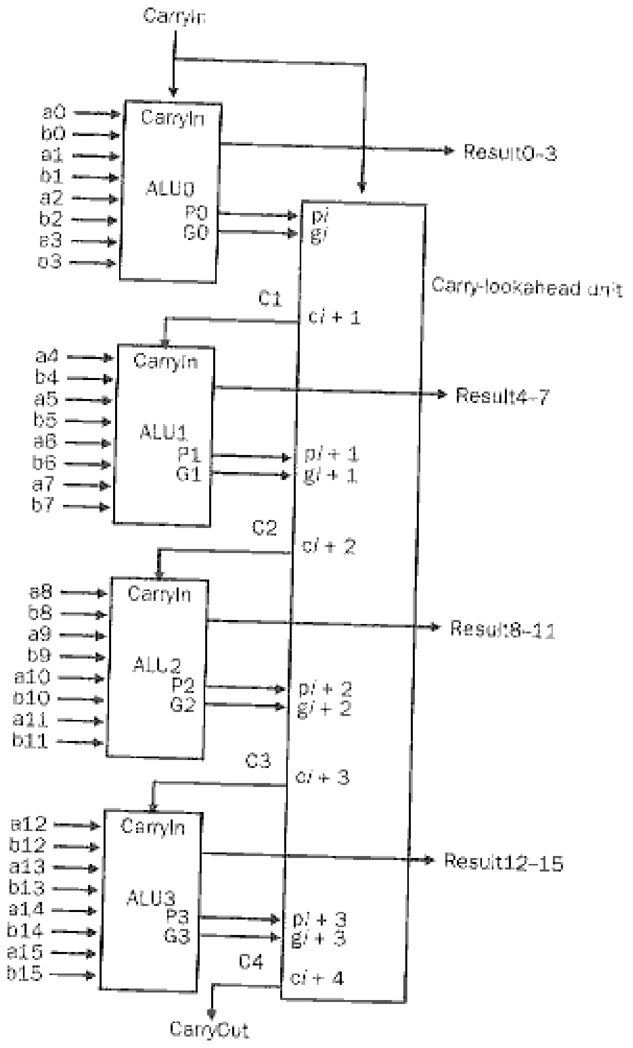


FIGURE 4.24 Four 4-bit ALUs using carry lookahead to form a 16-bit adder. Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

Then the equations at this higher level of abstraction for the carry in for each 4-bit group of the 16-bit adder (C1, C2, C3, C4 in Figure 4.24) are very similar to the carry out equations for each bit of the 4-bit adder (c1, c2, 3, c4) on page 243:

C1 =

C2 =

C3 =

C4 =

Figure 4.24 show
Exercises 4.44 thr
schemes, differen
the design of a 64

Both Levels

Example Determine the

a:
b:

Also, what is C

Answer Aligning the b
propagate pi (

a:
b:
gi:
pi:

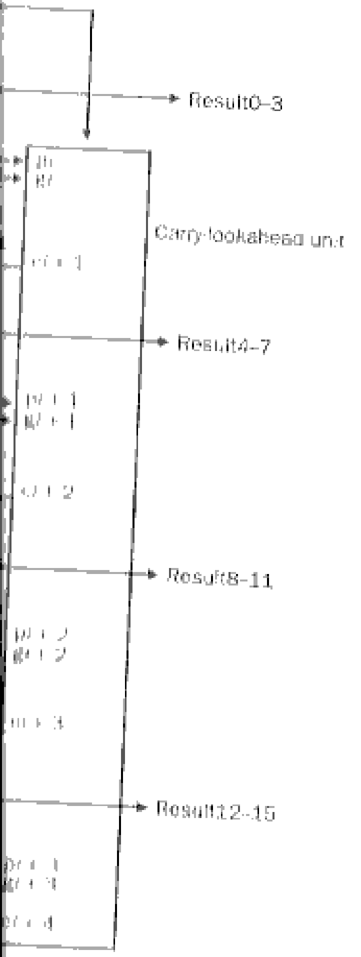
where the bits
propagates (P3
gates:

P3 = 1 · 1 ·

P2 = 1 · 1 ·

P1 = 1 · 1 ·

P0 = 1 · 0 ·



okahead to form a 16-bit adder. Note that the from the 4-bit ALUs

el of abstraction for the carry in for each C3, C4 in Figure 4.24) are very similar of the 4-bit adder (c1, c2, 3, c4) on page

$$C1 = G0 + (P0 \cdot c0)$$

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$

$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

Figure 4.24 shows 4-bit adders connected with such a carry lookahead unit. Exercises 4.44 through 4.48 explore the speed differences between these carry schemes, different notations for multibit propagate and generate signals, and the design of a 64-bit adder.

Both Levels of the Propagate and Generate

Example

Determine the *gi*, *pi*, *Pi*, and *Gi* values of these two 16-bit numbers:

```
a:        0001 1010 0011 0011two
b:        1110 0101 1110 1011two
```

Also, what is CarryOut15 (C4)?

Answer

Aligning the bits makes it easy to see the values of generate *gi* (*ai* · b*i*) and propagate *pi* (*ai* + b*i*):

```
a:        0001 1010 0011 0011
b:        1110 0101 1110 1011
gi:       0000 0000 0010 0011
pi:       1111 1111 1111 1011
```

where the bits are numbered 15 to 0 from left to right. Next, the "super" propagates (P3, P2, P1, P0) are simply the AND of the lower-level propagates:

$$P3 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P2 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0$$