Suppose you wanted to add the operation NOT (a AND b), called NAND. How could the ALU change to support it?

1. No change. You can calculate NAND quickly using the current ALU since \((a \cdot b) = \bar{a} + \bar{b}\) and we already have NOT a, NOT b, and OR.

2. You must expand the big multiplexor to add another input, and then add new logic to calculate NAND.

**Faster Addition: Carry Lookahead**

The key to speeding up addition is determining the carry in to the high-order bits sooner. 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**

As we mentioned earlier, 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

\[
\text{CarryIn}_2 = (b_1 \cdot \text{CarryIn}_1) + (a_1 \cdot \text{CarryIn}_1) + (a_1 \cdot b_1)
\]

Similarly, CarryIn1 is defined as

\[
\text{CarryIn}_1 = (b_0 \cdot \text{CarryIn}_0) + (a_0 \cdot \text{CarryIn}_0) + (a_0 \cdot b_0)
\]

Using the shorter and more traditional abbreviation of \(c_i\) for CarryIn\(_i\), we can rewrite the formulas as

\[
c_2 = (b_1 \cdot c_1) + (a_1 \cdot c_1) + (a_1 \cdot b_1)
\]

\[
c_1 = (b_0 \cdot c_0) + (a_0 \cdot c_0) + (a_0 \cdot b_0)
\]

Substituting the definition of \(c_1\) for the first equation results in this formula:

\[
c_2 = (a_1 \cdot a_0 \cdot b_0) + (a_1 \cdot a_0 \cdot c_0) + (a_1 \cdot b_0 \cdot c_0) + (b_1 \cdot a_0 \cdot c_0) + (b_1 \cdot b_0 \cdot c_0) + (a_1 \cdot b_1)
\]
You can imagine how the equation expands as we get to higher bits in the adder; it grows rapidly 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:

\[
c_{i+1} = (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i)
\]

\[
= (a_i \cdot b_i) + (a_i + b_i) \cdot c_i
\]

If we were to rewrite the equation for \(c_2\) using this formula, we would see some repeated patterns:

\[
c_2 = (a_1 \cdot b_1) + (a_1 + b_1) \cdot ((a_0 \cdot b_0) + (a_0 + b_0) \cdot c_0)
\]

Note the repeated appearance of \((a_i \cdot b_i)\) and \((a_i + b_i)\) in the formula above. These two important factors are traditionally called *generate* \((g_i)\) and *propagate* \((p_i)\):

\[
g_i = a_i \cdot b_i
\]

\[
p_i = a_i + b_i
\]

Using them to define \(c_{i+1}\), we get

\[
c_{i+1} = g_i + p_i \cdot c_i
\]

To see where the signals get their names, suppose \(g_i\) is 1. Then

\[
c_{i+1} = g_i + p_i \cdot c_i = 1 + p_i \cdot c_i = 1
\]

That is, the adder *generates* a CarryOut \((c_{i+1})\) independent of the value of CarryIn \((c_i)\). Now suppose that \(g_i\) is 0 and \(p_i\) is 1. Then

\[
c_{i+1} = g_i + p_i \cdot c_i = 0 + 1 \cdot c_i = c_i
\]

That is, the adder *propagates* CarryIn to a CarryOut. Putting the two together, CarryIn\(i+1\) is a 1 if either \(g_i\) is 1 or both \(p_i\) is 1 and CarryIn 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:

\[
\begin{align*}
c_1 &= g_0 + (p_0 \cdot c_0) \\
c_2 &= g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot c_0) \\
c_3 &= g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot c_0) \\
c_4 &= g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) + (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0)
\end{align*}
\]

These equations just represent common sense: CarryIn is a 1 if some earlier adder generates a carry and all intermediary adders propagate a carry. Figure B.6.1 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:

\[
\begin{align*}
P_0 &= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \\
P_1 &= p_7 \cdot p_6 \cdot p_5 \cdot p_4 \\
P_2 &= p_{11} \cdot p_{10} \cdot p_9 \cdot p_8 \\
P_3 &= p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}
\end{align*}
\]

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

For the “super” generate signal \( G_i \), 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:

\[
\begin{align*}
G_0 &= g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\
G_1 &= g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4) \\
G_2 &= g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8) \\
G_3 &= g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12})
\end{align*}
\]
FIGURE B.6.1 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 ($c_{i+1}$) will be full if either the nearest generate value ($g_i$) is turned on or if the $i$ propagate value ($p_i$) is on and there is water further upstream, either from an earlier generate, or propagate with water behind it. CarryIn ($c_0$) can result in a carry out without the help of any generates, but with the help of all propagates.
Figure B.6.2 updates our plumbing analogy to show P0 and G0.

**FIGURE B.6.2** 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.
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 B.6.3) are very similar to the carry out equations for each bit of the 4-bit adder (c1, c2, c3, c4) on page B-40:

\[
\begin{align*}
C1 &= G0 + (P0 \cdot c0) \\
C2 &= G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0) \\
C3 &= G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0) \\
C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)
\end{align*}
\]

Figure B.6.3 shows 4-bit adders connected with such a carry-lookahead unit. The exercises 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**

Determine the \( g_i \), \( p_i \), \( P_i \), and \( G_i \) values of these two 16-bit numbers:

\[
\begin{align*}
a: & \quad 0001 1010 0011 0011_{\text{two}} \\
b: & \quad 1110 0101 1110 1011_{\text{two}}
\end{align*}
\]

Also, what is CarryOut15 (C4)?

Aligning the bits makes it easy to see the values of generate \( g_i \) (ai \cdot bi) and propagate \( p_i \) (ai + bi):

\[
\begin{align*}
a: & \quad 0001 1010 0011 0011 \\
b: & \quad 1110 0101 1110 1011 \\
g_i: & \quad 0000 0000 0010 0011 \\
p_i: & \quad 1111 1111 1111 1011
\end{align*}
\]

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:

\[
\begin{align*}
P3 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\
P2 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\
P1 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\
P0 &= 1 \cdot 0 \cdot 1 \cdot 1 = 0
\end{align*}
\]
FIGURE B.6.3 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.
The “super” generates are more complex, so use the following equations:

\[
\begin{align*}
G_0 &= g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\
&= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0 \\
G_1 &= g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4) \\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1 \\
G_2 &= g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8) \\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \\
G_3 &= g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}) \\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \\
\end{align*}
\]

Finally, CarryOut15 is

\[
\begin{align*}
C_4 &= G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) \\
&\quad + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0) \\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\
&= 0 + 0 + 1 + 0 + 0 = 1
\end{align*}
\]

Hence there is a carry out when adding these two 16-bit numbers.

The reason carry lookahead can make carries faster is that all logic begins evaluating the moment the clock cycle begins, and the result will not change once the output of each gate stops changing. By taking a shortcut of going through fewer gates to send the carry in signal, the output of the gates will stop changing sooner, and hence the time for the adder can be less.

To appreciate the importance of carry lookahead, we need to calculate the relative performance between it and ripple carry adders.

### Speed of Ripple Carry versus Carry Lookahead

One simple way to model time for logic is to assume each AND or OR gate takes the same time for a signal to pass through it. Time is estimated by simply counting the number of gates along the path through a piece of logic. Compare the number of gate delays for paths of two 16-bit adders, one using ripple carry and one using two-level carry lookahead.

Figure B.5.5 on page B-28 shows that the carry out signal takes two gate delays per bit. Then the number of gate delays between a carry in to the least significant bit and the carry out of the most significant is \(16 \times 2 = 32\).
FIGURE B.5.5 Adder hardware for the carry out signal. The rest of the adder hardware is the logic for the Sum output given in the equation on page B-28.
For carry lookahead, the carry out of the most significant bit is just C4, defined in the example. It takes two levels of logic to specify C4 in terms of Pi and Gi (the OR of several AND terms). Pi is specified in one level of logic (AND) using pi, and Gi is specified in two levels using pi and gi, so the worst case for this next level of abstraction is two levels of logic. pi and gi are each one level of logic, defined in terms of ai and bi. If we assume one gate delay for each level of logic in these equations, the worst case is \(2 + 2 + 1 = 5\) gate delays.

Hence, for the path from carry in to carry out, the 16-bit addition by a carry-lookahead adder is six times faster, using this very simple estimate of hardware speed.

**Summary**

Carry lookahead offers a faster path than waiting for the carries to ripple through all 32 1-bit adders. This faster path is paved by two signals, generate and propagate. The former creates a carry regardless of the carry input, and the other passes a carry along. Carry lookahead also gives another example of how abstraction is important in computer design to cope with complexity.

**Check Yourself**

Using the simple estimate of hardware speed above with gate delays, what is the relative performance of a ripple carry 8-bit add versus a 64-bit add using carry-lookahead logic?

1. A 64-bit carry-lookahead adder is three times faster: 8-bit adds are 16 gate delays and 64-bit adds are 7 gate delays.
2. They are about the same speed, since 64-bit adds need more levels of logic in the 16-bit adder.
3. 8-bit adds are faster than 64 bits, even with carry lookahead.

**Elaboration:** We have now accounted for all but one of the arithmetic and logical operations for the core MIPS instruction set: the ALU in Figure B.5.14 omits support of shift instructions. It would be possible to widen the ALU multiplexor to include a left shift by 1 bit or a right shift by 1 bit. But hardware designers have created a circuit called a **barrel shifter**, which can shift from 1 to 31 bits in no more time than it takes to add two 32-bit numbers, so shifting is normally done outside the ALU.

**Elaboration:** The logic equation for the Sum output of the full adder on page B-28 can be expressed more simply by using a more powerful gate than AND and OR. An exclusive *OR* gate is true if the two operands disagree; that is,

\[x \neq y \Rightarrow 1 \text{ and } x == y \Rightarrow 0\]
In some technologies, exclusive OR is more efficient than two levels of AND and OR gates. Using the symbol ⊕ to represent exclusive OR, here is the new equation:

\[
\text{Sum} = a \oplus b \oplus \text{CarryIn}
\]

Also, we have drawn the ALU the traditional way, using gates. Computers are designed today in CMOS transistors, which are basically switches. CMOS ALU and barrel shifters take advantage of these switches and have many fewer multiplexors than shown in our designs, but the design principles are similar.

**Elaboration:** Using lowercase and uppercase to distinguish the hierarchy of generate and propagate symbols breaks down when you have more than two levels. An alternate notation that scales is \( g_{i..j} \) and \( p_{i..j} \) for the generate and propagate signals for bits \( i \) to \( j \). Thus, \( g_{1..1} \) is generate for bit 1, \( g_{4..1} \) is for bits 4 to 1, and \( g_{16..1} \) is for bits 16 to 1.

### B.7 Clocks

Before we discuss memory elements and sequential logic, it is useful to discuss briefly the topic of clocks. This short section introduces the topic and is similar to the discussion found in Section 5.2. More details on clocking and timing methodologies are presented in Section B.11.

Clocks are needed in sequential logic to decide when an element that contains state should be updated. A clock is simply a free-running signal with a fixed cycle time; the clock frequency is simply the inverse of the cycle time. As shown in Figure B.7.1, the clock cycle time or clock period is divided into two portions: when the clock is high and when the clock is low. In this text, we use only edge-triggered clocking. This means that all state changes occur on a clock edge. We use an edge-triggered methodology because it is simpler to explain. Depending on the technology, it may or may not be the best choice for a clocking methodology.

**edge-triggered clocking** A clocking scheme in which all state changes occur on a clock edge.

**clocking methodology** The approach used to determine when data is valid and stable relative to the clock.

![Clock signal oscillates between high and low values.](image)

**FIGURE B.7.1** A clock signal oscillates between high and low values. The clock period is the time for one full cycle. In an edge-triggered design, either the rising or falling edge of the clock is active and causes state to be changed.
A thorough book on logic design using Verilog.

A general text on logic design.

A general text on logic design.

**B.14 Exercises**

**B.1** [10] <§B.2> In More Depth: DeMorgan’s Theorems.


**B.3** [10] <§B.2> For More Practice: Truth Tables

**B.4** [10] <§B.2> For More Practice: Truth Tables

**B.5** [15] <§B.2> For More Practice: Building Logic Gates


**B.7** [10] <§§B.2, B.3> Construct the truth table for a four-input odd-parity function (see page B-65 for a description of parity).

**B.8** [10] <§§B.2, B.3> Implement the four-input odd-parity function with AND and OR gates using bubbled inputs and outputs.

**B.9** [10] <§§B.2, B.3> Implement the four-input odd-parity function with a PLA.

**B.10** [15] <§§B.2, B.3> Prove that a two-input multiplexor is also universal by showing how to build the NAND (or NOR) gate using a multiplexor.

**B.11** [5] <§§4.2, B.2, B.3> Assume that X consists of 3 bits, x2 x1 x0. Write four logic functions that are true if and only if

- X contains only one 0
- X contains an even number of 0s
- X when interpreted as an unsigned binary number is less than 4
- X when interpreted as a signed (two’s complement) number is negative

**B.12** [5] <§§4.2, B.2, B.3> Implement the four functions described in Exercise B.11 using a PLA.
B.13 [5] <§§4.2, B.2, B.3> Assume that X consists of 3 bits, x2 x1 x0, and Y consists of 3 bits, y2 y1 y0. Write logic functions that are true if and only if

- X < Y, where X and Y are thought of as unsigned binary numbers
- X < Y, where X and Y are thought of as signed (two’s complement) numbers
- X = Y

Use a hierarchical approach that can be extended to larger numbers of bits. Show how can you extend it to 6-bit comparison.

B.14 [5] <§§B.2, B.3> Implement a switching network that has two data inputs (A and B), two data outputs (C and D), and a control input (S). If S equals 1, the network is in pass-through mode, and C should equal A, and D should equal B. If S equals 0, the network is in crossing mode, and C should equal B, and D should equal A.


B.18 [5] <§§5.9, B.3> What is the function implemented by the following Verilog modules:

```verilog
module FUNC1 (I0, I1, S, out);
    input I0, I1;
    input S;
    output out;
    out = S? I1: I0;
endmodule

module FUNC2 (out,ctl,clk,reset);
    output [7:0] out;
    input ctl, clk, reset;
    reg [7:0] out;
    always @(posedge clk)
        if (reset) begin
            out <= 8'b0 ;
        end
        else if (ctl) begin
            out <= out + 1;
        end
        else begin
            out <= out - 1;
        end
endmodule
```

B.20 [10] <§§5.9, B.3, B.4> Write down a Verilog module implementation of a 2-to-4 decoder (and/or encoder).

B.21 [10] <§§5.9, B.3, B.4> Given the following logic diagram for an accumulator, write down the Verilog module implementation of it. Assume a positive edge-triggered register and asynchronous Rst.

B.22 [20] <§§4.5, B.3, B.4, B.5> Section 3.4 presents basic operation and possible implementations of multipliers. A basic unit of such implementations is a shift-and-add unit. Show a Verilog implementation for this unit. Show how can you use this unit to build a 32-bit multiplier.

B.23 [20] <§§4.6, B.3, B.4, B.5> Repeat Exercise B.22, but for an unsigned divider rather than a multiplier.

B.24 [15] <§B.5> The ALU supported set on less than (slt) using just the sign bit of the adder. Let’s try a set on less than operation using the values $-7_{ten}$ and $6_{ten}$. To make it simpler to follow the example, let’s limit the binary representations to 4 bits: $1001_{two}$ and $0110_{two}$.

$$1001_{two} - 0110_{two} = 1001_{two} + 1010_{two} = 0011_{two}$$

This result would suggest that $-7 > 6$, which is clearly wrong. Hence, we must factor in overflow in the decision.Modify the 1-bit ALU in Figure B.5.10 below to handle slt correctly. Make your changes on a photocopy of this figure to save time.
FIGURE B.5.10 (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or \( \bar{b} \), and (bottom) a 1-bit ALU for the most significant bit. The top drawing includes a direct input that is connected to perform the set on less than operation (see Figure B.5.11); the bottom has a direct output from the adder for the less than comparison called Set. (See Exercise 3.24 to see how to calculate overflow with fewer inputs.)
B.25 [20] <§B.6> A simple check for overflow during addition is to see if the CarryIn to the most significant bit is *not* the same as the CarryOut of the most significant bit. Prove that this check is the same as in Figure 3.3 on page 172.

B.26 [5] <§B.6> Rewrite the equations on page B-43 for a carry-lookahead logic for a 16-bit adder using a new notation. First, use the names for the CarryIn signals of the individual bits of the adder. That is, use c4, c8, c12, . . . instead of C1, C2, C3, . . . In addition, let \( p_{i,j} \) mean a propagate signal for bits \( i \) to \( j \), and \( g_{i,j} \) mean a generate signal for bits \( i \) to \( j \). For example, the equation

\[
C_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot c_0)
\]

can be rewritten as

\[
c_8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c_0)
\]

This more general notation is useful in creating wider adders.

B.27 [15] <§B.6> Write the equations for the carry-lookahead logic for a 64-bit adder using the new notation from Exercise B.26 and using 16-bit adders as building blocks. Include a drawing similar to Figure B.6.3 in your solution.

B.28 [10] <§B.6> Now calculate the relative performance of adders. Assume that hardware corresponding to any equation containing only OR or AND terms, such as the equations for \( p_i \) and \( g_i \) on page B-39, takes one time unit \( T \). Equations that consist of the OR of several AND terms, such as the equations for \( c_1, c_2, c_3, \) and \( c_4 \) on page B-40, would thus take two time units, \( 2T \). The reason is it would take \( T \) to produce the AND terms and then an additional \( T \) to produce the result of the OR. Calculate the numbers and performance ratio for 4-bit adders for both ripple carry and carry lookahead. If the terms in equations are further defined by other equations, then add the appropriate delays for those intermediate equations, and continue recursively until the actual input bits of the adder are used in an equation. Include a drawing of each adder labeled with the calculated delays and the path of the worst-case delay highlighted.

B.29 [15] <§B.6> This exercise is similar to Exercise B.28, but this time calculate the relative speeds of a 16-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, and the carry-lookahead scheme on page B-38.

B.30 [15] <§B.6> This exercise is similar to Exercises B.28 and B.29, but this time calculate the relative speeds of a 64-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, ripple carry of 16-bit groups that use carry lookahead, and the carry-lookahead scheme from Exercise B.27.
B.31 [10] <§B.6> In More Depth: Carry Save Adders

B.32 [10] <§B.6> In More Depth: Carry Save Adders

3.33 [10] <§B.6> There are times when we want to add a collection of numbers together. Suppose you wanted to add four 4-bit numbers (A, B, E, F) using 1-bit full adders. Let’s ignore carry lookahead for now. You would likely connect the 1-bit adders in the organization at the top of Figure B.14.1. Below the traditional organization is a novel organization of full adders. Try adding four numbers using both organizations to convince yourself that you get the same answer.

3.34 [5] <§B.6> First, show the block organization of the 16-bit carry save adders to add these 16 terms, as shown in Figure B.14.1. Assume that the time delay through each 1-bit adder is 2T. Calculate the time of adding four 4-bit numbers to the organization at the top versus the organization at the bottom of Figure B.14.1.
FIGURE B.14.1 Traditional ripple carry and carry save addition of four 4-bit numbers. The details are shown on the left, with the individual signals in lowercase, and the corresponding higher-level blocks are on the right, with collective signals in uppercase. Note that the sum of four $n$-bit numbers can take $n+2$ bits.

B.37  [10] <§B.10> For More Practice: Finite State Machines
B.38  [10] <§B.10> For More Practice: Finite State Machines
B.39  [15] <§§B.2, B.8, B.10> For More Practice: Constructing a Counter
B.40  [20] <§B.10> For More Practice: Constructing a Counter


§B.2, page B-8: No. If $A = 1$, $C = 1$, $B = 0$, the first is true, but the second is false.
§B.3, page B-20: c.
§B.4, page B-22: They are all exactly the same.
§B.4, page B-26: $A = 0$, $B = 1$.
§B.5, page B-38: 2.
§B.8, page B-57: c.
§B.10, page B-72: b.
§B.11, page B-77: No.