The field is in the lower 16 bits of the word and we want 0s in the upper bits of the result of the `andi`. In general, a shift left of 32 – (*n* + *m*) followed by a shift right by 32 – *n* will isolate any *n*-bit field whose least significant bit is in bit *m*.

Since `addi` and `slti` are intended for signed numbers, it is not surprising that their immediate fields are sign-extended before use. Branch and data transfer address fields are sign-extended as well.

Perhaps it *is* surprising that `addiu` and `sltiu` also sign-extend their immediates, but they do. The u stands for unsigned, but in reality `addiu` is often used simply as an `add` instruction that cannot overflow, and hence we often want to add negative numbers. It's much harder to come up for an excuse that `sltiu` does not sign extend its immediate.

Since `andi` and `ori` normally work with unsigned integers, the immediates are treated as unsigned integers as well, meaning that they are expanded to 32 bits by padding with leading 0s instead of sign extension. Thus if the bit fields in the third line of the example above extended beyond the 16 least significant bits, the `andi` instruction would need a 32-bit constant to avoid clearing the upper portion of the fields.

The MIPS assembler creates 32-bit constants with the pair of instructions `lui` and `ori`; see Chapter 3, page 147 for an example of creating 32-bit constants using `lui` and `addi`.

4.5 Constructing an Arithmetic Logic Unit

*ALU n. [**A**rthritic **L**ogic **U**nit or (rare) **A**rithmetic **L**ogic **U**nit] A random-number generator supplied as standard with all computer systems.*

Stan Kelly-Bootle, *The Devil's DP Dictionary,* 1981

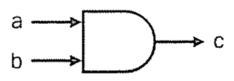
The *arithmetic logic unit* or *ALU* is the brawn of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. This section constructs an ALU from the four hardware building blocks shown in Figure 4.8 (see Appendix B for more details on these building blocks). Cases 1, 2, and 4 in Figure 4.8 all have two inputs. We will sometimes use versions of these components with more than two inputs, confident that you can generalize from this simple example. In any case, Appendix B provides examples with more inputs. (You may wish to review sections B.1 through B.3 before proceeding further.)

Because the MIPS word is 32 bits wide, we need a 32-bit-wide ALU. Let's assume that we will connect 32 1-bit ALUs to create the desired ALU. We'll therefore start by constructing a 1-bit ALU.

A 1-Bit ALU

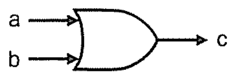
The logical operations are easiest, because they map directly onto the hardware components in Figure 4.8.

1. AND gate ($c = a \cdot b$)



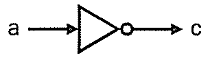
a	b	c = a · b
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate ($c = a + b$)



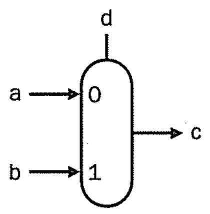
a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter ($c = \bar{a}$)



a	c = ā
0	1
1	0

4. Multiplexor
(if d = = 0, c = a;
else c = b)



d	c
0	a
1	b

FIGURE 4.8 Four hardware building blocks used to construct an arithmetic logic unit. The name of the operation and an equation describing it appear on the left. In the middle is the symbol for the block we will use in the drawings. On the right are tables that describe the outputs in terms of the inputs. Using the notation from Appendix B, a • b means "a AND b," a + b means "a OR b," and a line over the top (e.g., $\bar{a}$) means invert.

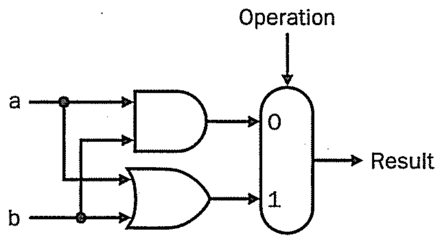


FIGURE 4.9 The 1-bit logical unit for AND and OR.

The 1-bit logical unit for AND and OR looks like Figure 4.9. The multiplexor on the right then selects *a* AND *b* or *a* OR *b*, depending on whether the value of *Operation* is 0 or 1. The line that controls the multiplexor is shown in color to distinguish it from the lines containing data. Notice that we have renamed the control and output lines of the multiplexor to give them names that reflect the function of the ALU.

The next function to include is addition. From Figure 4.3 on page 221 we can deduce the inputs and outputs of a single-bit adder. First, an adder must have two inputs for the operands and a single-bit output for the sum. There must be a second output to pass on the carry, called *CarryOut*. Since the CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called *CarryIn*. Figure 4.10 shows the inputs and the outputs of a 1-bit adder. Since we know what addition is supposed to do, we can specify the outputs of this "black box" based on its inputs, as Figure 4.11 demonstrates.

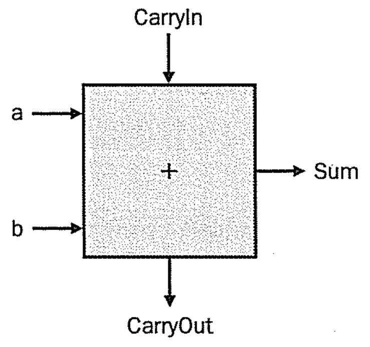


FIGURE 4.10 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2,2) adder or half adder.

Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

FIGURE 4.11 Input and output specification for a 1-bit adder.

From Appendix B, we know that we can express the output functions CarryOut and Sum as logical equations, and these equations can in turn be implemented with the building blocks in Figure 4.8. Let's do CarryOut. Figure 4.12 shows the values of the inputs when CarryOut is a 1.

We can turn this truth table into a logical equation, as explained in Appendix B. (Recall that a + b means "a OR b" and that a · b means "a AND b.")

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

If a · b · CarryIn is true, then one of the other three terms must also be true, so we can leave out this last term corresponding to the fourth line of the table. We can thus simplify the equation to

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

Figure 4.13 shows that the hardware within the adder black box for CarryOut consists of three AND gates and one OR gate. The three AND gates correspond exactly to the three parenthesized terms of the formula above for CarryOut, and the OR gate sums the three terms.

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

FIGURE 4.12 Values of the inputs when CarryOut is a 1.

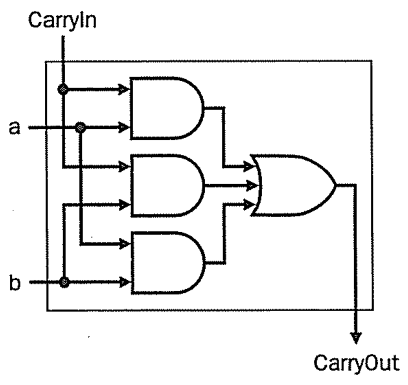


FIGURE 4.13 Adder hardware for the carry out signal. The rest of the adder hardware is the logic for the Sum output given in the equation above.

The Sum bit is set when exactly one input is 1 or when all three inputs are 1. The Sum results in a complex Boolean equation (recall that $\bar{a}$ means NOT a):

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

The drawing of the logic for the Sum bit in the adder black box is left as an exercise (see Exercise 4.43).

Figure 4.14 shows a 1-bit ALU derived by combining the adder with the earlier components. Sometimes designers also want the ALU to perform a few more simple operations, such as generating 0. The easiest way to add an operation is to expand the multiplexor controlled by the Operation line and, for this example, to connect 0 directly to the new input of that expanded multiplexor.

A 32-Bit ALU

Now that we have completed the 1-bit ALU, the full 32-bit ALU is created by connecting adjacent "black boxes." Using *xi* to mean the *i*th bit of *x*, Figure 4.15 shows a 32-bit ALU. Just as a single stone can cause ripples to radiate to the shores of a quiet lake, a single carry out of the least significant bit (Result0) can ripple all the way through the adder, causing a carry out of the most significant bit (Result31). Hence, the adder created by directly linking the carries of 1-bit adders is called a *ripple carry* adder. We'll see a faster way to connect the 1-bit adders starting on page 241.

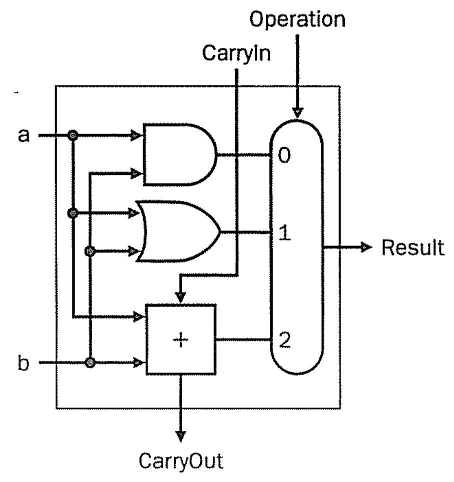


FIGURE 4.14 A 1-bit ALU that performs AND, OR, and addition (see Figure 4.13).

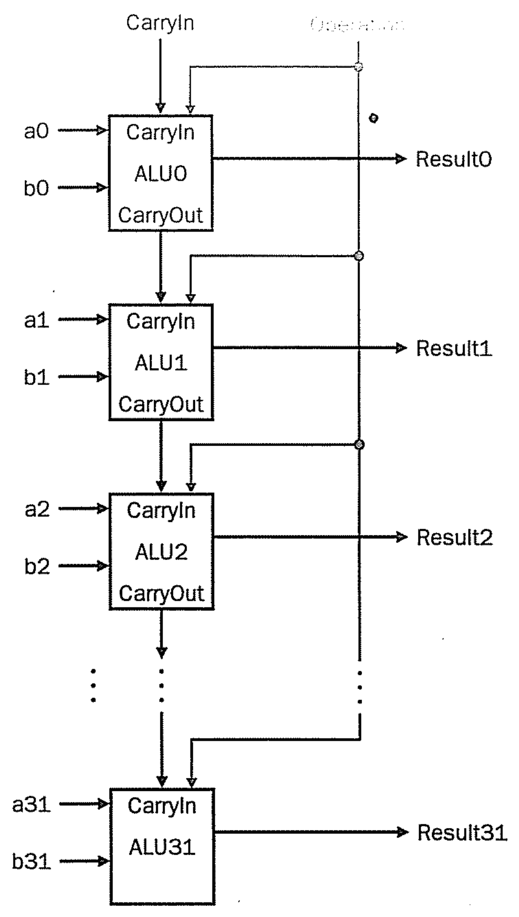


FIGURE 4.15 A 32-bit ALU constructed from 32 1-bit ALUs. CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

Subtraction is the same as adding the negative version of an operand, and this is how adders perform subtraction. Recall that the shortcut for negating a two's complement number is to invert each bit (sometimes called the *one's complement* as explained in the elaboration on page 219) and then add 1. To invert each bit, we simply add a 2:1 multiplexor that chooses between b and $\bar{b}$, as Figure 4.16 shows.

Suppose we connect 32 of these 1-bit ALUs, as we did in Figure 4.15. The added multiplexor gives the option of b or its inverted value, depending on Binvert, but this is only one step in negating a two's complement number. Notice that the least significant bit still has a CarryIn signal, even though it's unnecessary for addition. What happens if we set this CarryIn to 1 instead of 0?

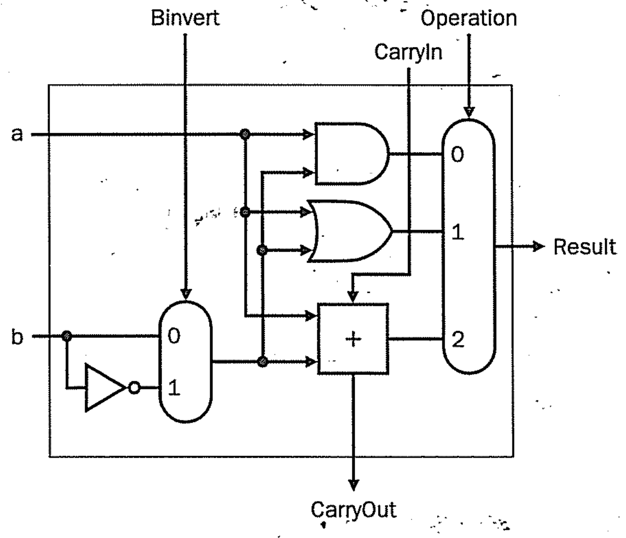


FIGURE 4.16 A 1-bit ALU that performs AND, OR, and addition on a and b or a and $\bar{b}$. By selecting b (Binvert = 1) and setting CarryIn to 1 in the least significant bit of the ALU, we get two's complement subtraction of b from a instead of addition of b to a.

The adder will then calculate a + b + 1. By selecting the inverted version of b, we get exactly what we want:

$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

The simplicity of the hardware design of a two's complement adder helps explain why two's complement representation has become the universal standard for integer computer arithmetic.

Tailoring the 32-Bit ALU to MIPS

This set of operations—add, subtract, AND, OR—is found in the ALU of almost every computer. If we look at Figure 4.7 on page 228, we see that the operations of most MIPS instructions can be performed by this ALU. But the design of the ALU is incomplete.

One instruction that still needs support is the set on less than instruction (`slt`). Recall that the operation produces 1 if rs < rt, and 0 otherwise. Consequently, `slt` will set all but the least significant bit to 0, with the least significant bit set according to the comparison. For the ALU to perform `slt`, we first need to expand the three-input multiplexor in Figure 4.16 to add an input for the `slt` result. We call that new input *Less*, and use it only for `slt`.

The top drawing of Figure 4.17 shows the new 1-bit ALU with the expanded multiplexor. From the description of `slt` above, we must connect 0 to the Less input for the upper 31 bits of the ALU, since those bits are always set to 0. What remains to consider is how to compare and set *the least significant bit* for set on less than instructions.

What happens if we subtract b from a? If the difference is negative, then a < b since

$$\begin{aligned}(a - b) < 0 &\Rightarrow ((a - b) + b) < (0 + b)\\ &\Rightarrow a < b\end{aligned}$$

We want the least significant bit of a set on less than operation to be a 1 if a < b; that is, a 1 if a – b is negative and a 0 if it's positive. This desired result corresponds exactly to the sign-bit values: 1 means negative and 0 means positive. Following this line of argument, we need only connect the sign bit from the adder output to the least significant bit to get set on less than.

Unfortunately, the Result output from the most significant ALU bit in the top of Figure 4.17 for the `slt` operation is *not* the output of the adder; the ALU output for the `slt` operation is obviously the input value Less.

Thus, we need a new 1-bit ALU for the most significant bit that has an extra output bit: the adder output. The bottom drawing of Figure 4.17 shows the design, with this new adder output line called *Set*, and used only for `slt`. As long as we need a special ALU for the most significant bit, we added the overflow detection logic since it is also associated with that bit.

Alas, the test of less than is a little more complicated than just described because of overflow; Exercise 4.23 on page 326 explores what must be done. Figure 4.18 shows the 32-bit ALU.

Notice that every time we want the ALU to subtract, we set both CarryIn and Binvert to 1. For adds or logical operations, we want both control lines to be 0. We can therefore simplify control of the ALU by combining the CarryIn and Binvert to a single control line called *Bnegate*.

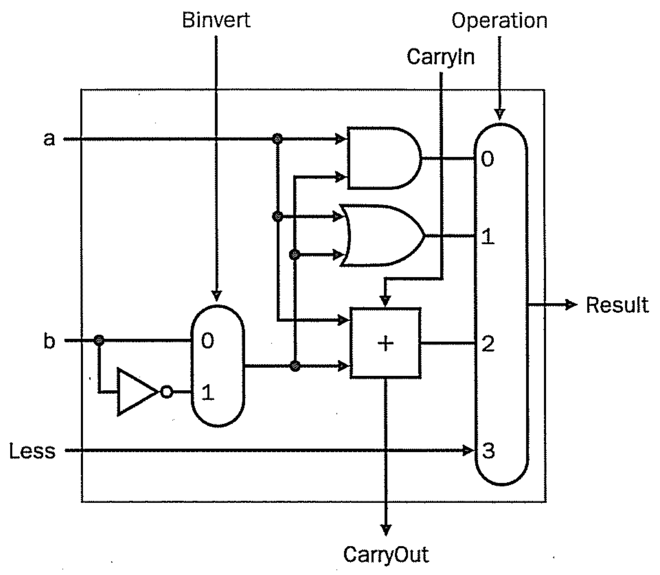
To further tailor the ALU to the MIPS instruction set, we must support conditional branch instructions. These instructions branch either if two registers are equal or if they are unequal. The easiest way to test equality with the ALU is to subtract b from a and then test to see if the result is 0 since

$$(a - b = 0) \Rightarrow a = b$$

Thus, if we add hardware to test if the result is 0, we can test for equality. The simplest way is to OR all the outputs together and then send that signal through an inverter:

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0})}$$

Figure 4.19 shows the revised 32-bit ALU. We can think of the combination of the 1-bit Bnegate line and the 2-bit Operation lines as 3-bit control lines for



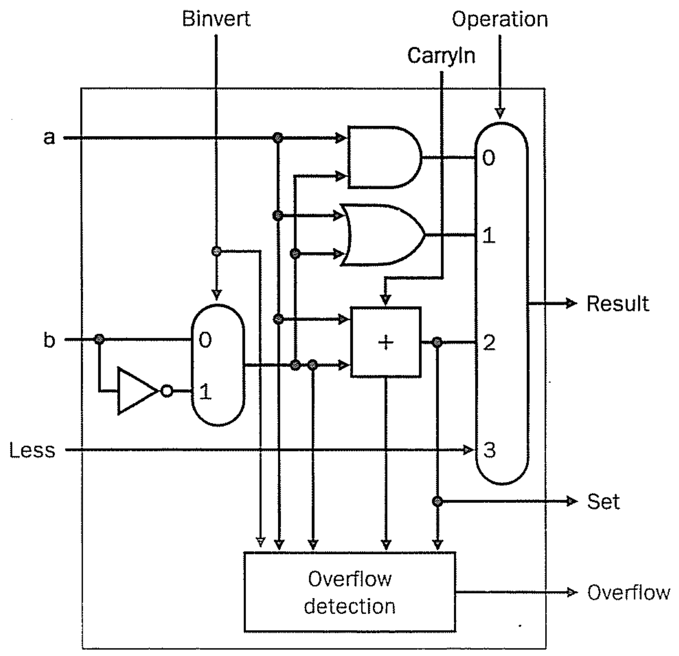


FIGURE 4.17 (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or $\overline{b}$, and (bottom) a 1-bit ALU for the most significant bit. The top drawing includes a direct input that is connected to perform the set on less than operation (see Figure 4.18); the bottom has a direct output from the adder for the less than comparison called Set. (Refer to Exercise 4.42 to see how to calculate overflow with fewer inputs.)

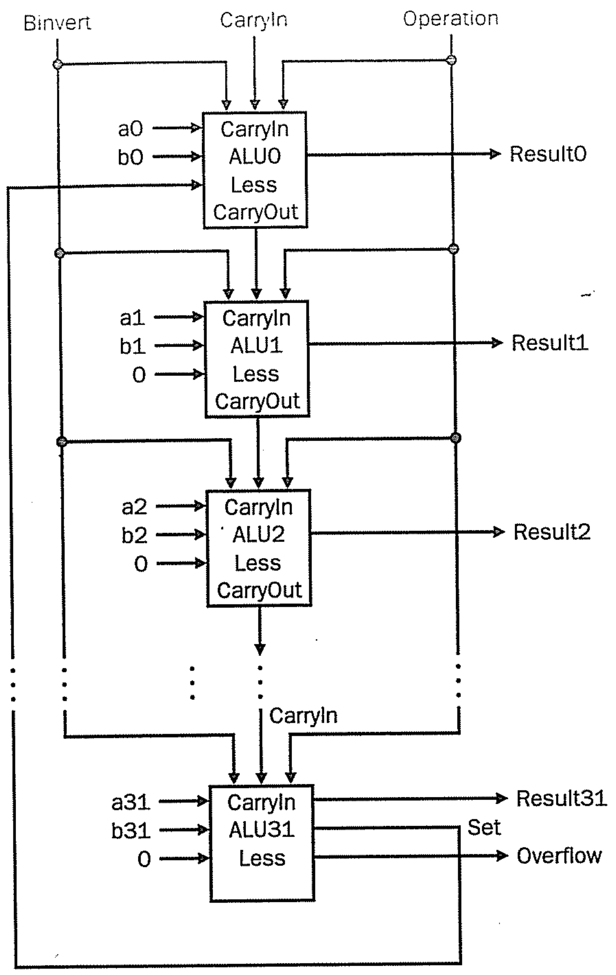


FIGURE 4.18 A 32-bit ALU constructed from the 31 copies of the 1-bit ALU in the top of Figure 4.17 and one 1-bit ALU in the bottom of that figure. The Less inputs are connected to 0 except for the least significant bit, and that is connected to the Set output of the most significant bit. If the ALU performs a – b and we select the input 3 in the multiplexor in Figure 4.17, then Result = 0 . . . 001 if a < b, and Result = 0 . . . 000 otherwise.

the ALU, telling it to perform add, subtract, AND, OR, or set on less than. Figure 4.20 shows the ALU control lines and the corresponding ALU operation.

Finally, now that we have seen what is inside a 32-bit ALU, we will use the universal symbol for a complete ALU, as shown in Figure 4.21.

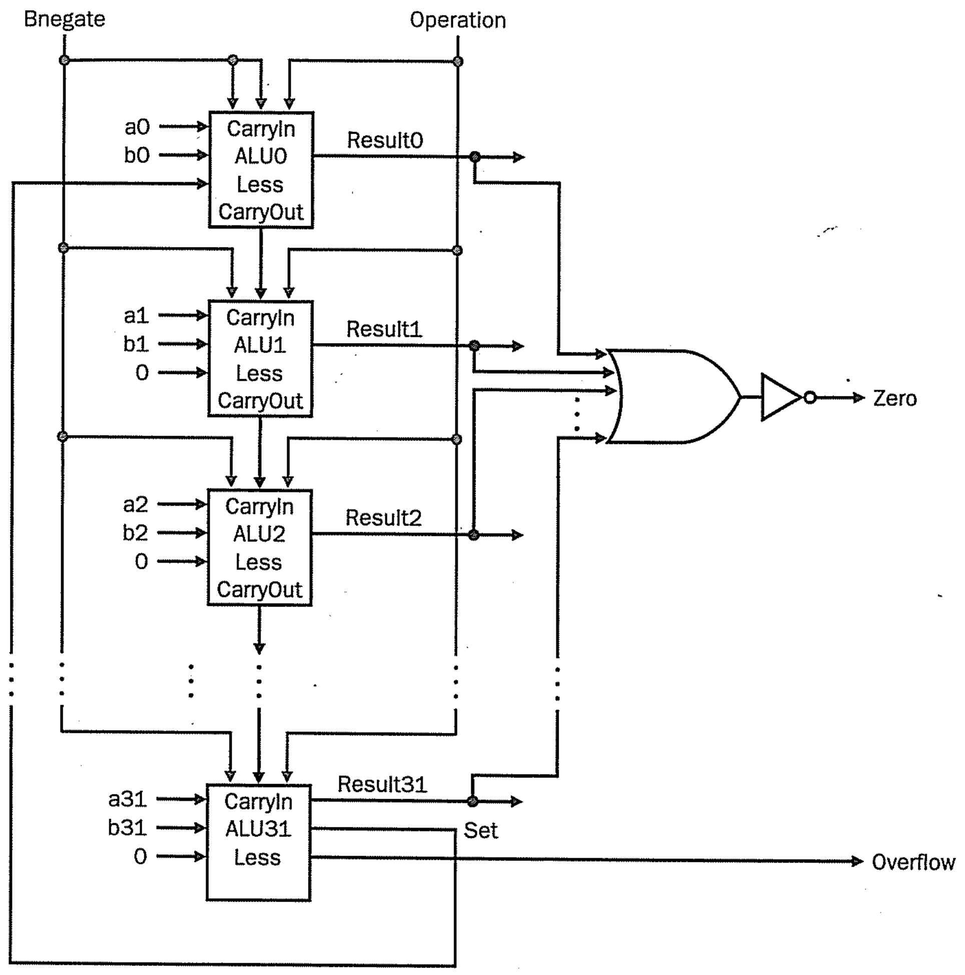


FIGURE 4.19 The final 32-bit ALU. This adds a Zero detector to Figure 4.18.

ALU control lines	Function
000	and
001	or
010	add
110	subtract
111	set on less than

FIGURE 4.20 The values of the three ALU control lines Bnegate and Operation and the corresponding ALU operations.

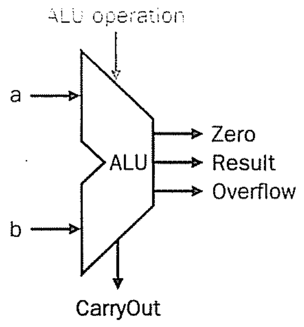


FIGURE 4.21 The symbol commonly used to represent an ALU, as shown in Figure 4.19. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

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

$$\begin{aligned} c2 &= (b1 \cdot c1) + (a1 \cdot c1) + (a1 \cdot b1) \\ c1 &= (b0 \cdot c0) + (a0 \cdot c0) + (a0 \cdot b0) \end{aligned}$$

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*):

$$\begin{aligned} gi &= ai \cdot bi \\ pi &= ai + bi \end{aligned}$$

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.

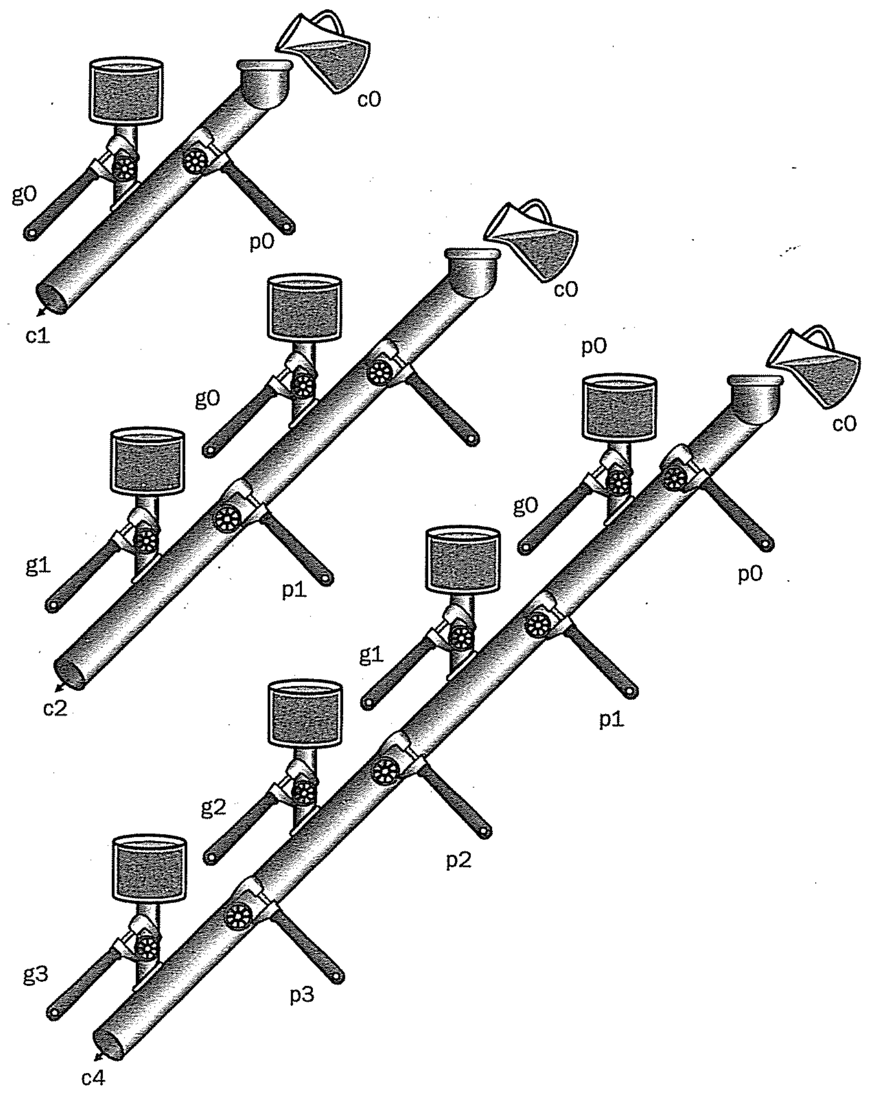


FIGURE 4.22 A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water, pipes, and valves. The wrenches are turned to open and close valves. Water is shown in color. The output of the pipe (c*i*+1) will be full if either the nearest generate value (g*i*) is turned on or if the *i* propagate value (p*i*) is on and there is water further upstream, either from an earlier generate, or propagate with water behind it. CarryIn (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 (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:

$$G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)$$

$$G1 = g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)$$

$$G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)$$

$$G3 = g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)$$

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

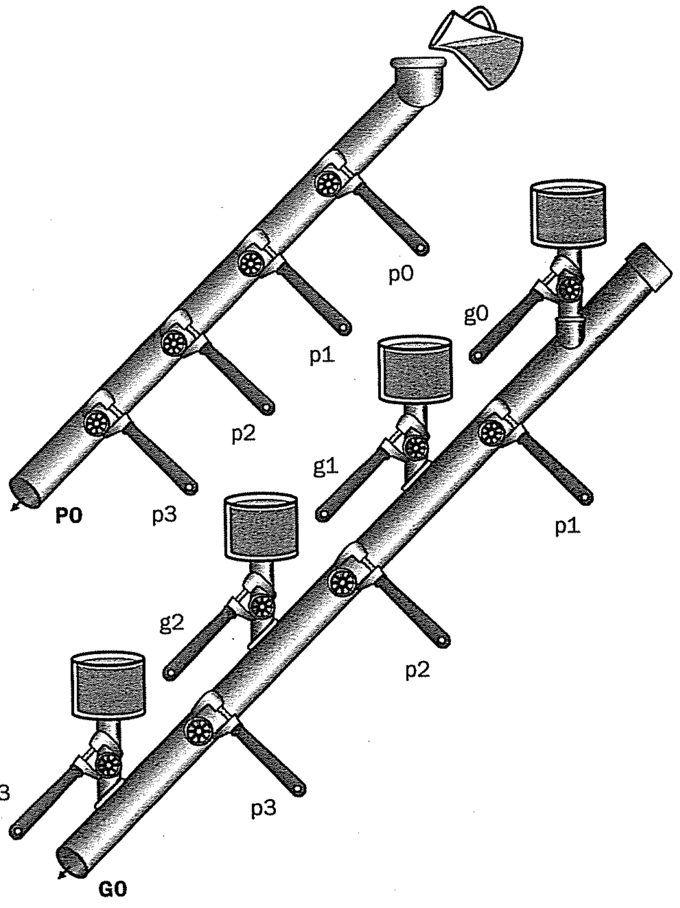


FIGURE 4.23 A plumbing analogy for the next-level carry-lookahead signals P0 and G0. P0 is open only if all four propagates (p*i*) are open, while water flows in G0 only if at least one generate (g*i*) is open and all the propagates downstream from that generate are open.

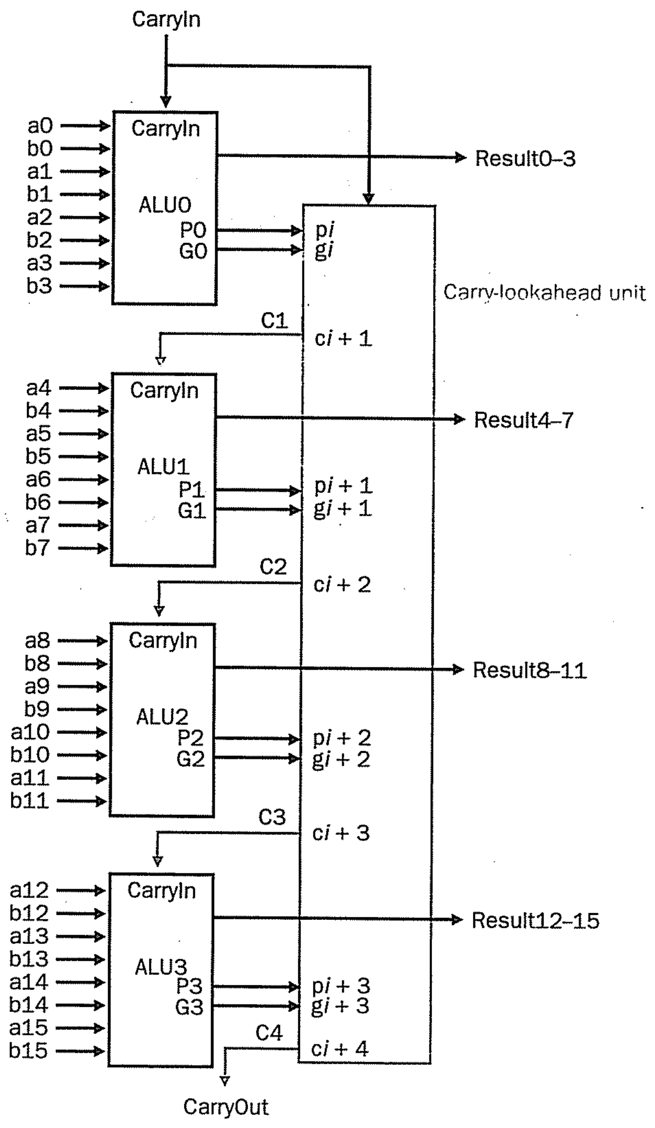


FIGURE 4.24 Four 4-bit ALUs using carry lookahead to form a 16-bit adder. Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

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:

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

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

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

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

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 0011_{two}
b: 1110 0101 1110 1011_{two}

Also, what is CarryOut15 (C4)?

Answer

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

```
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:

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

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

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

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

The "super" generates are more complex, so use the followinge quations:

$$
\begin{aligned}
G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)\\
&= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0\\
G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)\\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1\\
G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)\\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0\\
G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)\\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0
\end{aligned}
$$

Finally, CarryOut15 is

$$
\begin{aligned}
C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0)\\
&\qquad + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)\\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0)\\
&= 0 + 0 + 1 + 0 + 0 = 1
\end{aligned}
$$

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

Example

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 longest path through a piece of logic. Compare the number of *gate delays* for the critical paths of two 16-bit adders, one using ripple carry and one using two-level carry lookahead.

Answer

Figure 4.13 on page 233 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 $16 \times 2 = 32$.

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 P*i* and G*i* (the OR of several AND terms). P*i* is specified in one level of logic (AND) using p*i*, and G*i* is specified in two levels using p*i* and g*i*, so the worst case for this next level of abstraction is two levels of logic. p*i* and g*i* are each one level of logic, defined in terms of a*i* and b*i*. 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 16-bit addition a carry-lookahead adder is six times faster, using this simple estimate of hardware speed.

Summary

The primary point of this section is that the traditional ALU can be constructed from a multiplexor and a few gates that are replicated 32 times. To make it more useful to the MIPS architecture, we expand the traditional ALU with hardware to test if the result is 0, detect overflow, and perform the basic operation for set on less than.

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.

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 4.21 omits support of shift instructions. It would be possible to widen the ALU multiplexor to include a left shift by 1 bit or 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 234 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 $\oplus$ 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.