Check Yourself Assuming all values are initially zero, what are the values of A and B after executing this Verilog code inside an `always` block?

```
C=1;
A <= C;
B = C;
```

B.5 Constructing a Basic 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 (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 four hardware building blocks (AND and OR gates, inverters, and multiplexors) and illustrates how combinational logic works. In the next section, we will see how addition can be sped up through more clever designs.

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 B.2.1.

The 1-bit logical unit for AND and OR looks like Figure B.5.1. 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. 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 B.5.2 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 B.5.3 demonstrates.

We can express the output functions CarryOut and Sum as logical equations, and these equations can in turn be implemented with logic gates. Let's do CarryOut. Figure B.5.4 shows the values of the inputs when CarryOut is a 1.

We can turn this truth table into a logical equation:

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

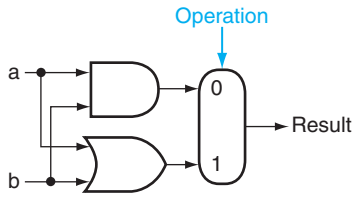

FIGURE B.5.1 The 1-bit logical unit for AND and OR.

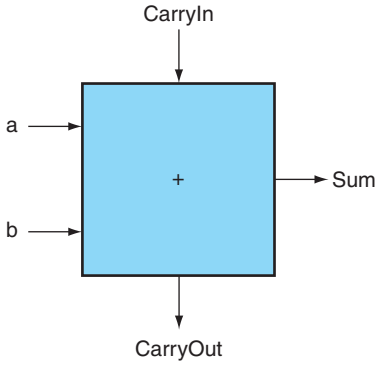

FIGURE B.5.2 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

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 B.5.3 Input and output specification for a 1-bit adder.

If a · b · CarryIn is true, then all 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 B.5.5 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 B.5.4 Values of the inputs when CarryOut is a 1.

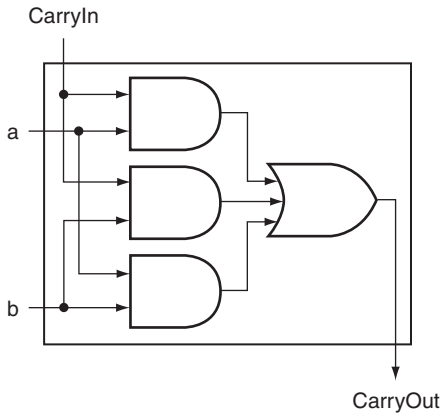

FIGURE B.5.5 Adder hardware for the CarryOut signal. The rest of the adder hardware is the logic for the Sum output given in the equation on this page.

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 $\overline{a}$ means NOT a):

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

The drawing of the logic for the Sum bit in the adder black box is left as an exercise for the reader.

Figure B.5.6 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.

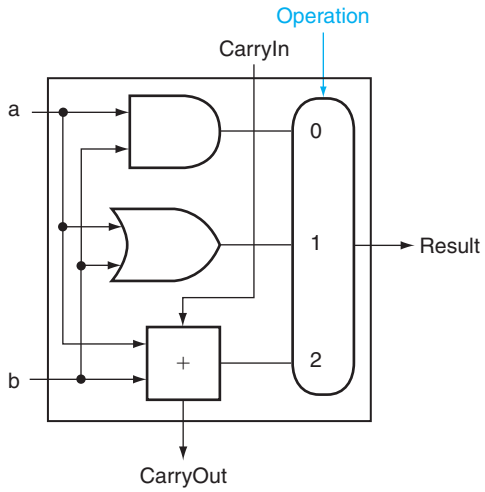

FIGURE B.5.6 A 1-bit ALU that performs AND, OR, and addition (see Figure B.5.5).

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 B.5.7 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 B-38.

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*) and then add 1. To invert each bit, we simply add a 2:1 multiplexor that chooses between b and $\overline{b}$, as Figure B.5.8 shows.

Suppose we connect 32 of these 1-bit ALUs, as we did in Figure B.5.7. The added multiplexor gives the option of b or its inverted value, depending on Binvert, but

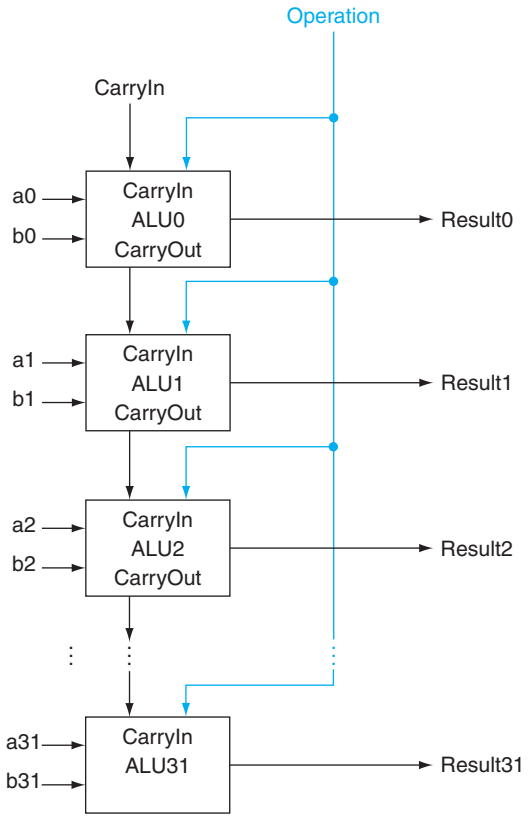

FIGURE B.5.7 A 32-bit ALU constructed from 32 1-bit ALUs. CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

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

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

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.

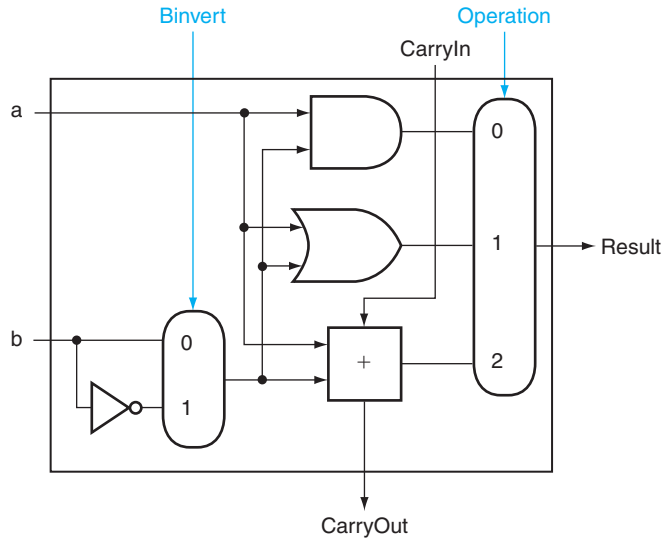

FIGURE B.5.8 A 1-bit ALU that performs AND, OR, and addition on a and b or a and $\overline{b}$. By selecting $\overline{b}$ (Binvert = 1) and setting CarryIn to 1 in the least significant bit of the ALU, we get two's complement subtraction of b from a instead of addition of b to a.

A MIPS ALU also needs a NOR function. Instead of adding a separate gate for NOR, we can reuse much of the hardware already in the ALU, like we did for subtract. The insight comes from the following truth about NOR:

$$\overline{(a + b)} = \overline{a} \cdot \overline{b}$$

That is, NOT (a OR b) is equivalent to NOT a AND NOT b. This fact is called DeMorgan's theorem and is explored in the exercises in more depth.

Since we have AND and NOT b, we only need to add NOT a to the ALU. Figure B.5.9 shows that change.

Tailoring the 32-Bit ALU to MIPS

These four operations—add, subtract, AND, OR—are found in the ALU of almost every computer, and 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

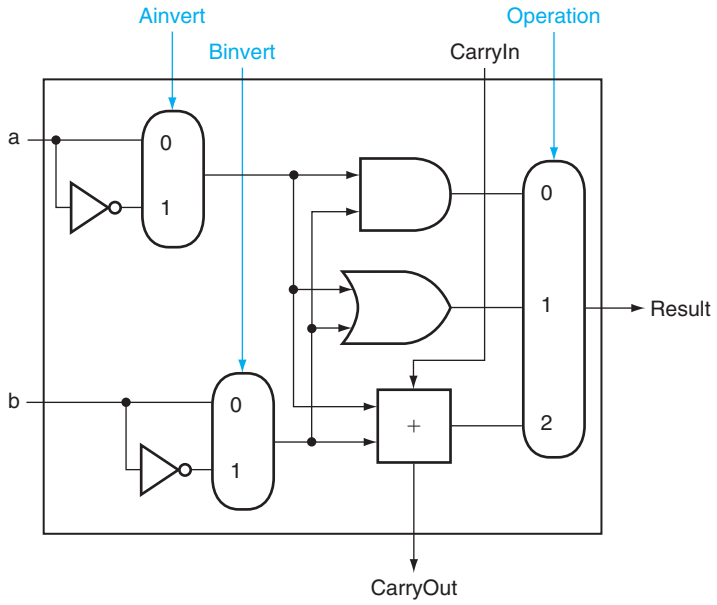

FIGURE B.5.9 A 1-bit ALU that performs AND, OR, and addition on a and b or $\overline{a}$ and $\overline{b}$. By selecting $\overline{a}$ (Ainvert = 1) and $\overline{b}$ (Binvert = 1), we get a NOR b instead of a AND b.

multiplexor in Figure B.5.8 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 B.5.10 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

$$(a - b) < 0 \Rightarrow ((a - b) + b) < (0 + b)$$
$$\Rightarrow a < b$$

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 B.5.10 for the `slt` operation is *not* the output of the adder; the ALU output for the `slt` operation is obviously the input value Less.

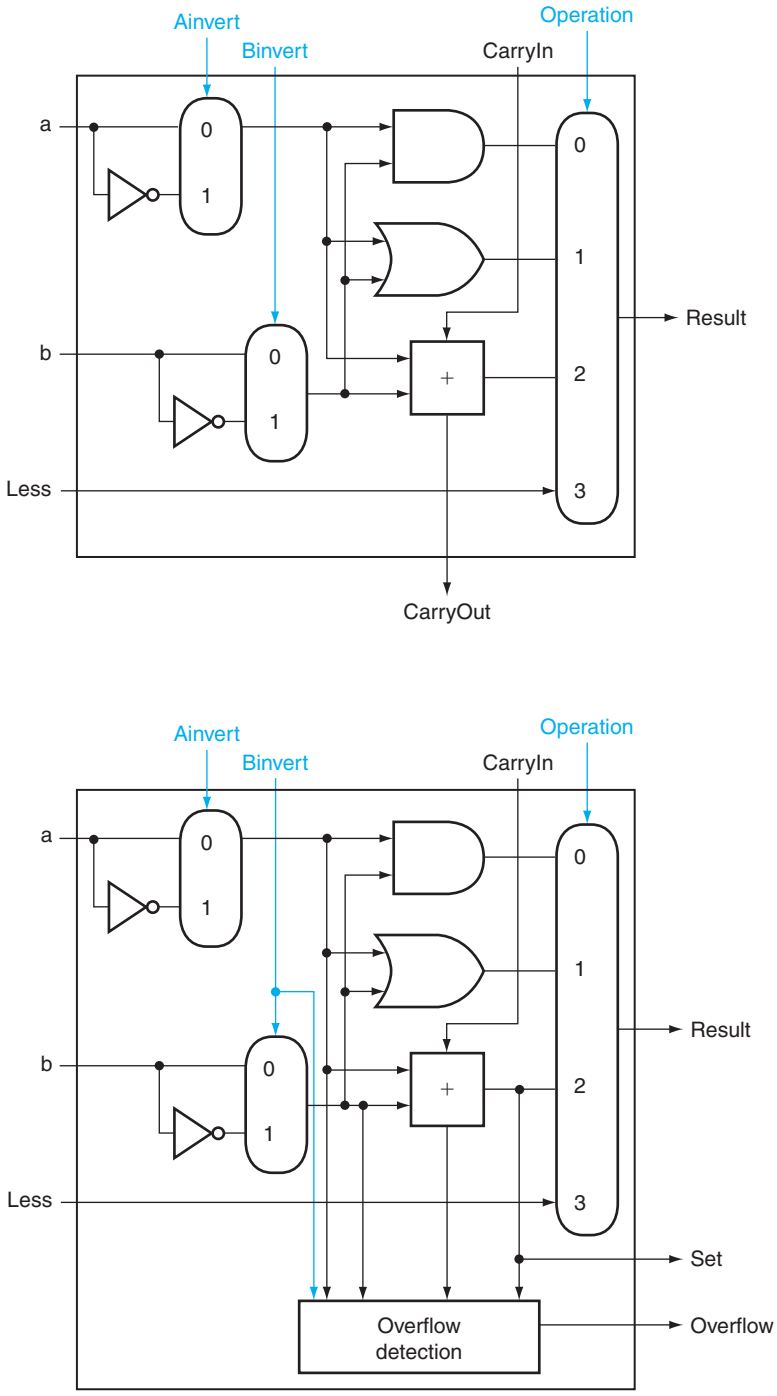

FIGURE B.5.10 (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or $\overline{b}$, and (bottom) a 1-bit ALU for the most significant bit. The top drawing includes a direct input that is connected to perform the set on less than operation (see Figure B.5.11); the bottom has a direct output from the adder for the less than comparison called Set. (See Exercise B.24 at the end of this appendix to see how to calculate overflow with fewer inputs.)

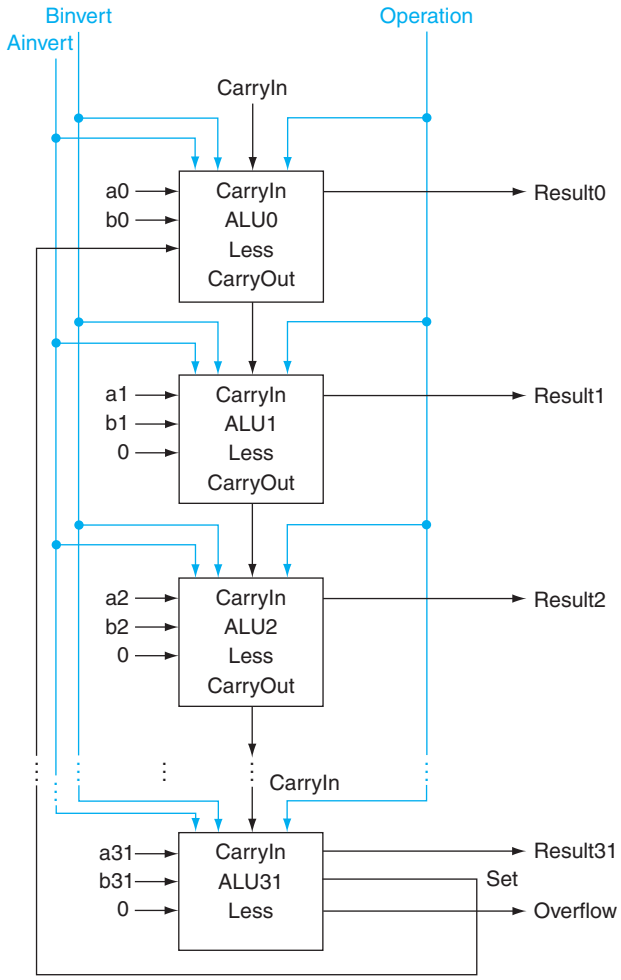

FIGURE B.5.11 A 32-bit ALU constructed from the 31 copies of the 1-bit ALU in the top of Figure B.5.10 and one 1-bit ALU in the bottom of that figure. The Less inputs are connected to 0 except for the least significant bit, which is connected to the Set output of the most significant bit. If the ALU performs a − b and we select the input 3 in the multiplexor in Figure B.5.10, then Result = 0 … 001 if a < b, and Result = 0 … 000 otherwise.

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 B.5.10 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, as we explore in the exercises. Figure B.5.11 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 B.5.12 shows the revised 32-bit ALU. We can think of the combination of the 1-bit Ainvert line, the 1-bit Binvert line, and the 2-bit Operation lines as 4-bit control lines for the ALU, telling it to perform add, subtract, AND, OR, or set on less than. Figure B.5.13 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 B.5.14.

Defining the MIPS ALU in Verilog

Figure B.5.15 shows how a combinational MIPS ALU might be specified in Verilog; such a specification would probably be compiled using a standard parts library that provided an adder, which could be instantiated. For completeness, we show the ALU control for MIPS in Figure B.5.16, which is used in Chapter 4, where we build a Verilog version of the MIPS datapath.

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. The next section explores how to speed-up addition. This topic is not crucial to understanding the rest of the appendix and may be skipped.

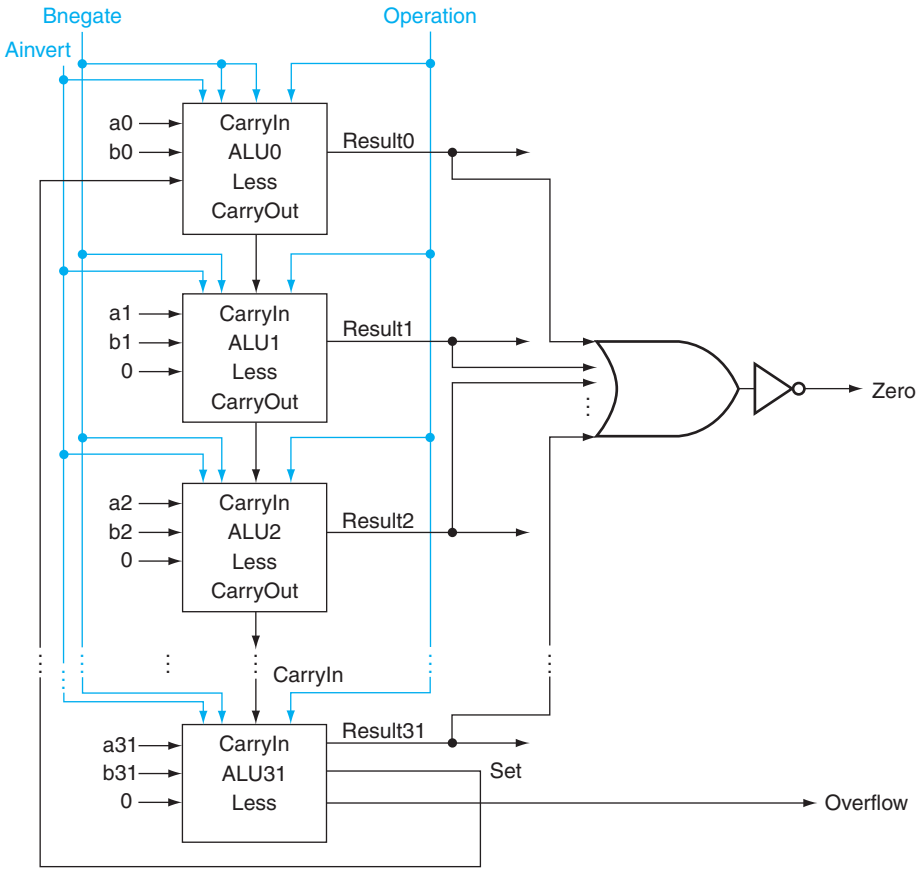

FIGURE B.5.12 The final 32-bit ALU. This adds a Zero detector to Figure B.5.11.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

FIGURE B.5.13 The values of the three ALU control lines, Bnegate, and Operation, and the corresponding ALU operations.

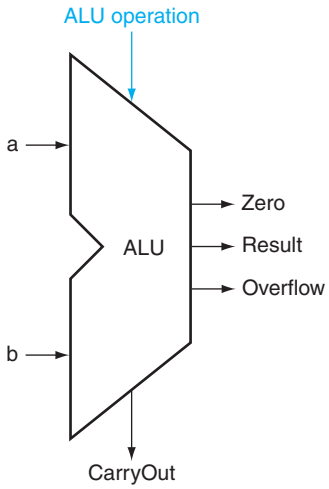

FIGURE B.5.14 The symbol commonly used to represent an ALU, as shown in Figure B.5.12. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
    end
endmodule
```

FIGURE B.5.15 A Verilog behavioral definition of a MIPS ALU.