2 Boolean Arithmetic

Counting is the religion of this generation, its hope and salvation.
—Gertrude Stein (1874–1946)

In this chapter we build gate logic designs that represent numbers and perform arithmetic operations on them. Our starting point is the set of logic gates built in chapter 1, and our ending point is a fully functional Arithmetic Logical Unit. The ALU is the centerpiece chip that executes all the arithmetic and logical operations performed by the computer. Hence, building the ALU functionality is an important step toward understanding how the Central Processing Unit (CPU) and the overall computer work.

As usual, we approach this task gradually. The first section gives a brief Background on how binary codes and Boolean arithmetic can be used, respectively, to represent and add signed numbers. The Specification section describes a succession of *adder chips*, designed to add two bits, three bits, and pairs of *n*-bit binary numbers. This sets the stage for the ALU specification, which is based on a sophisticated yet simple logic design. The Implementation and Project sections provide tips and guidelines on how to build the adder chips and the ALU on a personal computer, using the hardware simulator supplied with the book.

Binary addition is a simple operation that runs deep. Remarkably, most of the operations performed by digital computers can be reduced to elementary additions of binary numbers. Therefore, constructive understanding of binary addition holds the key to the implementation of numerous computer operations that depend on it, one way or another.

2.1   Background

Binary Numbers   Unlike the decimal system, which is founded on base 10, the binary system is founded on base 2. When we are given a certain binary pattern, say "10011," and we are told that this pattern is supposed to represent an integer number, the decimal value of this number is computed by convention as follows:

$$(10011)_{two} = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19 \tag{1}$$

In general, let $x = x_n x_{n-1} \ldots x_0$ be a string of digits. The *value* of x in base b, denoted $(x)_b$, is defined as follows:

$$(x_n x_{n-1} \ldots x_0)_b = \sum_{i=0}^{n} x_i \cdot b^i \tag{2}$$

The reader can verify that in the case of $(10011)_{two}$, rule (2) reduces to calculation (1).

The result of calculation (1) happens to be 19. Thus, when we press the keyboard keys labeled '1', '9' and ENTER while running, say, a spreadsheet program, what ends up in some register in the computer's memory is the binary code 10011. More precisely, if the computer happens to be a 32-bit machine, what gets stored in the register is the bit pattern 00000000000000000000000000010011.

Binary Addition   A pair of binary numbers can be added digit by digit from right to left, according to the same elementary school method used in decimal addition. First, we add the two right-most digits, also called the *least significant bits* (**LSB**) of the two binary numbers. Next, we add the resulting carry bit (which is either 0 or 1) to the sum of the next pair of bits up the significance ladder. We continue the process until the two *most significant bits* (**MSB**) are added. If the last bit-wise addition generates a carry of 1, we can report overflow; otherwise, the addition completes successfully:

0	0	0	1		(carry)	1	1	1	1
	1	**0**	**0**	**1**	x		**1**	**0**	**1**	**1**
+	**0**	**1**	**0**	**1**	y	+	**0**	**1**	**1**	**1**
0	1	1	1	0	$x + y$	1	0	0	1	0
	no overflow						overflow			

We see that computer hardware for binary addition of two n-bit numbers can be built from logic gates designed to calculate the sum of three bits (pair of bits plus carry bit). The transfer of the resulting carry bit forward to the addition of the next significant pair of bits can be easily accomplished by proper wiring of the 3-bit adder gates.

Signed Binary Numbers   A binary system with n digits can generate a set of 2^n different bit patterns. If we have to represent signed numbers in binary code, a natural solution is to split this space into two equal subsets. One subset of codes is assigned to represent positive numbers, and the other negative numbers. Ideally, the coding scheme should be chosen in such a way that the introduction of signed numbers would complicate the hardware implementation as little as possible.

This challenge has led to the development of several coding schemes for representing signed numbers in binary code. The method used today by almost all computers is called the *2's complement* method, also known as *radix complement*. In a binary system with n digits, the 2's complement of the number x is defined as follows:

$$\bar{x} = \begin{cases} 2^n - x & if \ x \neq 0 \\ 0 & otherwise \end{cases}$$

For example, in a 5-bit binary system, the 2's complement representation of -2 or "minus$(00010)_{two}$" is $2^5 - (00010)_{two} = (32)_{ten} - (2)_{ten} = (30)_{ten} = (11110)_{two}$. To check the calculation, the reader can verify that $(00010)_{two} + (11110)_{two} = (00000)_{two}$. Note that in the latter computation, the sum is actually $(100000)_{two}$, but since we are dealing with a 5-bit binary system, the left-most sixth bit is simply ignored. As a rule, when the 2's complement method is applied to n-bit numbers, $x + (-x)$ always sums up to 2^n (i.e., 1 followed by n 0's)—a property that gives the method its name. Figure 2.1 illustrates a 4-bit binary system with the 2's complement method.

An inspection of figure 2.1 suggests that an n-bit binary system with 2's complement representation has the following properties:

Positive numbers		Negative numbers	
0	0000		
1	0001	1111	−1
2	0010	1110	−2
3	0011	1101	−3
4	0100	1100	−4
5	0101	1011	−5
6	0110	1010	−6
7	0111	1001	−7
		1000	−8

Figure 2.1   2's complement representation of signed numbers in a 4-bit binary system.

- The system can code a total of 2^n signed numbers, of which the maximal and minimal numbers are $2^{n-1} - 1$ and -2^{n-1}, respectively.

- The codes of all positive numbers begin with a 0.

- The codes of all negative numbers begin with a 1.

- To obtain the code of $-x$ from the code of x, leave all the trailing (least significant) 0's and the first least significant 1 intact, then flip all the remaining bits (convert 0's to 1's and vice versa). An equivalent shortcut, which is easier to implement in hardware, is to flip all the bits of x and add 1 to the result.

A particularly attractive feature of this representation is that addition of any two signed numbers in 2's complement is exactly the same as addition of positive numbers. Consider, for example, the addition operation $(-2) + (-3)$. Using 2's complement (in a 4-bit representation), we have to add, in binary, $(1110)_{two} + (1101)_{two}$. Without paying any attention to which numbers (positive or negative) these codes represent, bit-wise addition will yield 1011 (after throwing away the overflow bit). As figure 2.1 shows, this indeed is the 2's complement representation of -5.

In short, we see that the 2's complement method facilitates the addition of any two signed numbers without requiring special hardware beyond that needed for simple bit-wise addition. What about subtraction? Recall that in the 2's complement method, the arithmetic negation of a signed number x, that is, computing $-x$, is achieved by negating all the bits of x and adding 1 to the result. Thus subtraction can be easily handled by $x - y = x + (-y)$. Once again, hardware complexity is kept to a minimum.

The material implications of these theoretical results are significant. Basically, they imply that a single chip, called *Arithmetic Logical Unit*, can be used to encapsulate all the basic arithmetic and logical operators performed in hardware. We now turn to specify one such ALU, beginning with the specification of an *adder* chip.
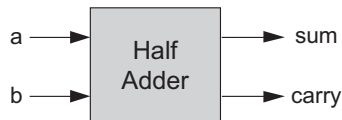
2.2   Specification

2.2.1   Adders

We present a hierarchy of three adders, leading to a multi-bit adder chip:

- *Half-adder:*   designed to add two bits
- *Full-adder:*   designed to add three bits
- *Adder:*   designed to add two n-bit numbers

Inputs		Outputs	
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

```
Chip name: HalfAdder
Inputs:    a, b
Outputs:   sum, carry
Function:  sum   = LSB of a + b
           carry = MSB of a + b
```

Figure 2.2   Half-adder, designed to add 2 bits.

We also present a special-purpose adder, called *incrementer*, designed to add 1 to a given number.

Half-Adder   The first step on our way to adding binary numbers is to be able to add two bits. Let us call the least significant bit of the addition `sum`, and the most significant bit `carry`. Figure 2.2 presents a chip, called half-adder, designed to carry out this operation.

Full-Adder   Now that we know how to add two bits, figure 2.3 presents a *full-adder* chip, designed to add three bits. Like the half-adder case, the full-adder chip produces two outputs: the least significant bit of the addition, and the carry bit.

Adder   Memory and register chips represent integer numbers by *n*-bit patterns, *n* being 16, 32, 64, and so forth—depending on the computer platform. The chip whose job is to add such numbers is called a multi-bit adder, or simply *adder*. Figure 2.4 presents a 16-bit adder, noting that the same logic and specifications scale up as is to any *n*-bit adder.

Incrementer   It is convenient to have a special-purpose chip dedicated to adding the constant 1 to a given number. Here is the specification of a 16-bit incrementer:
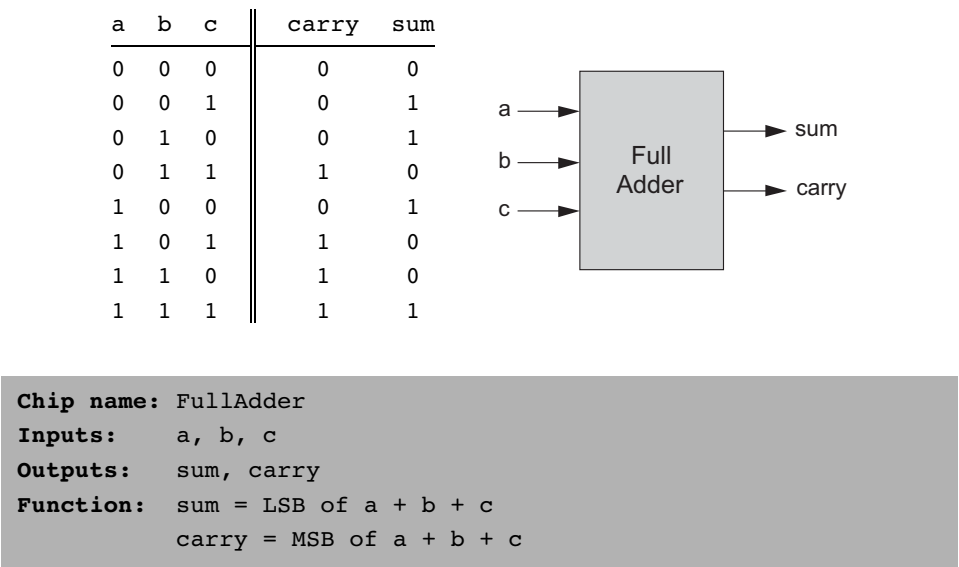
a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1


```
Chip name: FullAdder
Inputs:    a, b, c
Outputs:   sum, carry
Function:  sum = LSB of a + b + c
           carry = MSB of a + b + c
```

Figure 2.3   Full-adder, designed to add 3 bits.

```
... 1 0 1 1 a
... 0 0 1 0 b    +
_____
... 1 1 0 1 out
```



```
Chip name: Add16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  out = a + b
Comment:   Integer 2's complement addition.
           Overflow is neither detected nor handled.
```
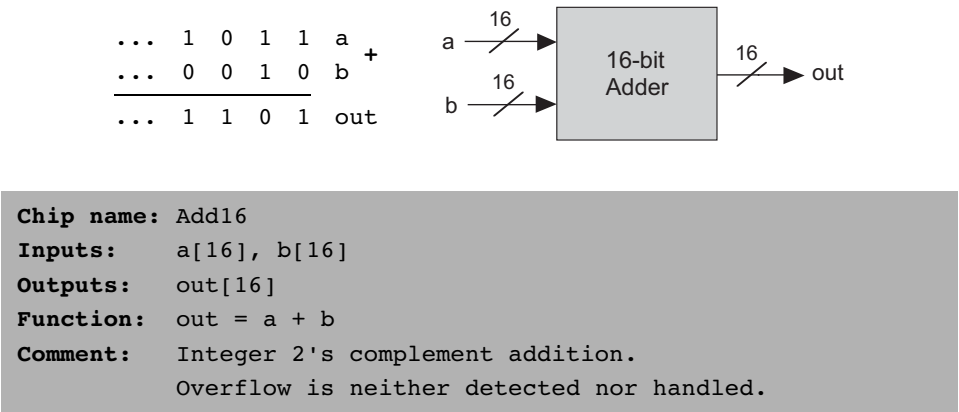
Figure 2.4   16-bit adder. Addition of two *n*-bit binary numbers for any *n* is ''more of the same.''

```
Chip name: Inc16
Inputs:     in[16]
Outputs:    out[16]
Function:   out=in+1
Comment:    Integer 2's complement addition.
            Overflow is neither detected nor handled.
```

2.2.2   The Arithmetic Logic Unit (ALU)

The specifications of the adder chips presented so far were generic, meaning that they hold for any computer. In contrast, this section describes an ALU that will later become the centerpiece of a specific computer platform called *Hack*. At the same time, the principles underlying the design of our ALU are rather general. Further, our ALU architecture achieves a great deal of functionality using a minimal set of internal parts. In that respect, it provides a good example of an efficient and elegant logic design.

The Hack ALU computes a fixed set of functions $out = f_i(x, y)$ where x and y are the chip's two 16-bit inputs, *out* is the chip's 16-bit output, and f_i is an arithmetic or logical function selected from a fixed repertoire of eighteen possible functions. We instruct the ALU which function to compute by setting six input bits, called *control bits*, to selected binary values. The exact input-output specification is given in figure 2.5, using pseudo-code.

Note that each one of the six control bits instructs the ALU to carry out a certain elementary operation. Taken together, the combined effects of these operations cause the ALU to compute a variety of useful functions. Since the overall operation is driven by six control bits, the ALU can potentially compute $2^6 = 64$ different functions. Eighteen of these functions are documented in figure 2.6.

We see that programming our ALU to compute a certain function $f(x, y)$ is done by setting the six control bits to the code of the desired function. From this point on, the internal ALU logic specified in figure 2.5 should cause the ALU to output the value $f(x, y)$ specified in figure 2.6. Of course, this does not happen miraculously, it's the result of careful design.

For example, let us consider the twelfth row of figure 2.6, where the ALU is instructed to compute the function x-1. The zx and nx bits are 0, so the x input is neither zeroed nor negated. The zy and ny bits are 1, so the y input is first zeroed, and then negated bit-wise. Bit-wise negation of zero, $(000\ldots00)_{two}$, gives $(111\ldots11)_{two}$, the 2's complement code of -1. Thus the ALU inputs end up being x
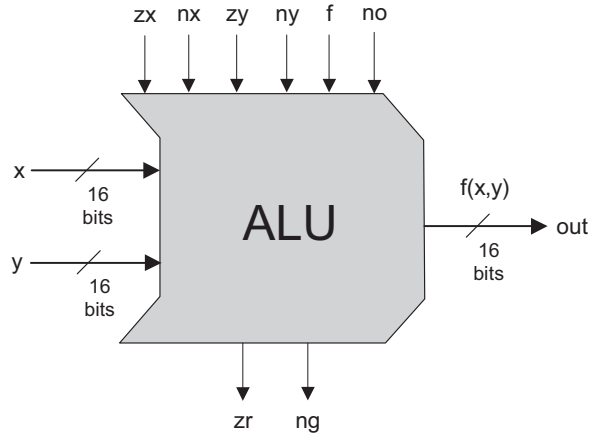
```
Chip name: ALU
Inputs:    x[16], y[16],    // Two 16-bit data inputs
           zx,              // Zero the x input
           nx,              // Negate the x input
           zy,              // Zero the y input
           ny,              // Negate the y input
           f,               // Function code: 1 for Add, 0 for And
           no               // Negate the out output
Outputs:   out[16],         // 16-bit output
           zr,              // True iff out=0
           ng               // True iff out<0
Function:  if zx then x = 0      // 16-bit zero constant
           if nx then x = !x     // Bit-wise negation
           if zy then y = 0      // 16-bit zero constant
           if ny then y = !y     // Bit-wise negation
           if f then out = x + y  // Integer 2's complement addition
               else out = x & y  // Bit-wise And
           if no then out = !out  // Bit-wise negation
           if out=0 then zr = 1 else zr = 0  // 16-bit eq. comparison
           if out<0 then ng = 1 else ng = 0  // 16-bit neg. comparison
Comment:   Overflow is neither detected nor handled.
```

Figure 2.5   The Arithmetic Logic Unit.

These bits instruct how to preset the x input		These bits instruct how to preset the y input		This bit selects between + / And	This bit inst. how to postset out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Figure 2.6 The ALU truth table. Taken together, the binary operations coded by the first six columns effect the function listed in the right column (we use the symbols !, &, and | to represent the operators Not, And, and Or, respectively, performed bit-wise). The complete ALU truth table consists of sixty-four rows, of which only the eighteen presented here are of interest.

and -1. Since the f-bit is 1, the selected operation is *arithmetic addition*, causing the ALU to calculate x+(-1). Finally, since the no bit is 0, the output is not negated but rather left as is. To conclude, the ALU ends up computing x-1, which was our goal.

Does the ALU logic described in figure 2.6 compute every one of the other seventeen functions listed in the figure's right column? To verify that this is indeed the case, the reader can pick up some other rows in the table and prove their respective ALU operation. We note that some of these computations, beginning with the

function $f(x, y) = 1$, are not trivial. We also note that there are some other useful functions computed by the ALU but not listed in the figure.

It may be instructive to describe the thought process that led to the design of this particular ALU. First, we made a list of all the primitive operations that we wanted our computer to be able to perform (right column in figure 2.6). Next, we used backward reasoning to figure out how x, y, and out can be manipulated in binary fashion in order to carry out the desired operations. These processing requirements, along with our objective to keep the ALU logic as simple as possible, have led to the design decision to use six control bits, each associated with a straightforward binary operation. The resulting ALU is simple and elegant. And in the hardware business, simplicity and elegance imply inexpensive and powerful computer systems.

2.3   Implementation

Our implementation guidelines are intentionally partial, since we want you to discover the actual chip architectures yourself. As usual, each chip can be implemented in more than one way; the simpler the implementation, the better.

Half-Adder   An inspection of figure 2.2 reveals that the functions $sum(a, b)$ and $carry(a, b)$ happen to be identical to the standard $Xor(a, b)$ and $And(a, b)$ Boolean functions. Thus, the implementation of this adder is straightforward, using previously built gates.

Full-Adder   A full adder chip can be implemented from two half adder chips and one additional simple gate. A direct implementation is also possible, without using half-adder chips.

Adder   The addition of two signed numbers represented by the 2's complement method as two n-bit buses can be done bit-wise, from right to left, in n steps. In step 0, the least significant pair of bits is added, and the carry bit is fed into the addition of the next significant pair of bits. The process continues until in step $n - 1$ the most significant pair of bits is added. Note that each step involves the addition of three bits. Hence, an n-bit adder can be implemented by creating an array of n full-adder chips and propagating the carry bits up the significance ladder.

Incrementer   An n-bit incrementer can be implemented trivially from an n-bit adder.

ALU   Note that our ALU was carefully planned to effect all the desired ALU operations *logically*, using simple Boolean operations. Therefore, the *physical* implementation of the ALU is reduced to implementing these simple Boolean operations, following their pseudo-code specifications. Your first step will likely be to create a logic circuit that manipulates a 16-bit input according to the `nx` and `zx` control bits (i.e., the circuit should conditionally zero and negate the 16-bit input). This logic can be used to manipulate the `x` and `y` inputs, as well as the `out` output. Chips for bitwise And-ing and addition have already been built in this and in the previous chapter. Thus, what remains is to build logic that chooses between them according to the `f` control bit. Finally, you will need to build logic that integrates all the other chips into the overall ALU. (When we say "build logic," we mean "write HDL code").

2.4   Perspective

The construction of the multi-bit adder presented in this chapter was standard, although no attention was paid to efficiency. In fact, our suggested adder implementation is rather inefficient, due to the long delays incurred while the carry bit propagates from the least significant bit pair to the most significant bit pair. This problem can be alleviated using logic circuits that effect so-called carry look-ahead techniques. Since addition is one of the most prevalent operations in any given hardware platform, any such low-level improvement can result in dramatic and global performance gains throughout the computer.

In any given computer, the overall functionality of the hardware/software platform is delivered jointly by the ALU and the operating system that runs on top of it. Thus, when designing a new computer system, the question of how much functionality the ALU should deliver is essentially a cost/performance issue. The general rule is that hardware implementations of arithmetic and logical operations are usually more costly, but achieve better performance. The design trade-off that we have chosen in this book is to specify an ALU hardware with a limited functionality and then implement as many operations as possible in software. For example, our ALU features neither multiplication nor division nor floating point arithmetic. We will implement some of these operations (as well as more mathematical functions) at the operating system level, described in chapter 12.

Detailed treatments of Boolean arithmetic and ALU design can be found in most computer architecture textbooks.

2.5   Project

Objective   Implement all the chips presented in this chapter. The only building blocks that you can use are the chips that you will gradually build and the chips described in the previous chapter.

Tip   When your HDL programs invoke chips that you may have built in the previous project, we recommend that you use the built-in versions of these chips instead. This will ensure correctness and speed up the operation of the hardware simulator. There is a simple way to accomplish this convention: Make sure that your project directory includes only the `.hdl` files that belong to the present project.

The remaining instructions for this project are identical to those of the project from the previous chapter, except that the last step should be replaced with "Build and simulate all the chips specified in the `projects/02` directory."