2. Boolean Arithmetic[1]

Counting is the religion of this generation, its hope and salvation.

(Gertrude Stein, American writer, 1874-1946)

In this chapter we build the Boolean circuits that represent numbers and perform arithmetic operations on them. Our starting point is the set of logic gates we built in the last chapter, 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.

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:

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

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

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

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 is the binary code 10011. More precisely, if the computer happens to be a 32-bit machine, say, 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* of the two binary numbers. Next, we add the 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* are added. If the last bit-wise addition generates a carry of 1, we can report overflow; otherwise, the addition completes successfully.

[1] From *The Digital Core*, by Nisan & Schocken, forthcoming in 2003, www.idc.ac.il/csd

0	0	0	1	0	(carry)	1	1	1	1	0
	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
valid: no overflow						overflow				

FIGURE 2-1: Two examples of binary addition (assuming 4-bit registers). In the case of overflow, the computation can be rendered invalid.

We see that computer hardware for binary addition must be able to calculate the sum of three bits (pair of bits plus carry bit) and pass the carry bit from the addition of one pair of bits to the addition of the next significant pair of bits.

Signed binary numbers: A binary system with *n* digits can code 2^n different bit patterns. Let us call this set of patterns the system's "code space". 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. The exact coding scheme should be chosen in such a way that, ideally, the introduction of negative 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 modern 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:

$$\bar{x} = \begin{cases} 2^n - x & if\ x \neq 0 \\ 0 & otherwise \end{cases} \qquad (3)$$

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 6th bit is lost. 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). This property gives the method its name. Table 2-2 illustrates a 4-bit binary system with the 2's complement method.

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

TABLE 2-2: the 2's complement representation of integer numbers, assuming a 4-bit binary system.

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

- 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 numbers in 2's complement is exactly identical to 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+1101. Without paying any attention to which numbers (positive or negative) these codes represent, bitwise addition will then yield 1011 (after throwing away the 5'th overflow bit). Indeed, this is the 2's complement representation of (-5).

In sum, we are able to perform addition of any two signed numbers without introducing any complexity to the underlying hardware beyond simple bitwise addition. What about subtraction? Recall that in the 2's complement method, the arithmetic negation of a signed number x, i.e. computing $-x$, is achieved by negating all the bits of x and adding 1 to the result. Thus subtraction can be handled by $x-y=x+(-y)$. Once again, hardware complexity is kept to a minimum.

The material implications of these theoretical results are quite significant: in most computer architectures today, a single chip, called *Arithmetic Logical Unit,* is used to encapsulate all the arithmetic and logical operators performed in hardware. This ALU -- the centerpiece of the computer's execution -- is dedicated to performing elementary operations such as adding, subtracting, ANDing and ORing binary bit patterns. We now turn to specify one such ALU, beginning with the specification of an adder chip.

2.2 Specification

Adders

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

- *Half-adder*: designed to add 2 bits;
- *Full-adder*: designed to add 3 bits;
- *Adder*: designed to add two *n*-bit numbers.

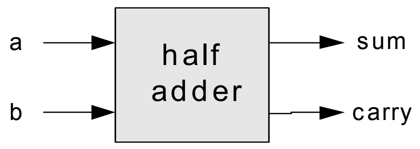
We also present a special-purpose adder, called *incrementer*, designed to add 1 to a given number.

Half Adder: The first step on our way to adding binary numbers is to be able to add two bits. This task requires the handling of four possible cases:

$$\begin{aligned} 0+0 &= 00 \\ 0+1 &= 01 \\ 1+0 &= 01 \\ 1+1 &= 10 \end{aligned}$$

We will now present a chip, called *half-adder*, that implements this addition operation. The least significant bit of the addition is called sum, and the most significant bit is called carry.

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

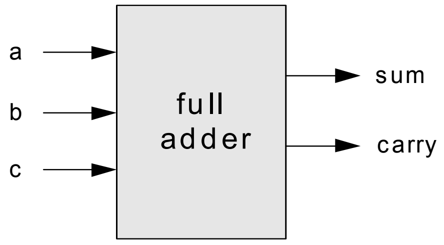


```
Chip name:  HalfAdder
Inputs:     a, b
Outputs:    sum, carry
Function:   sum   = LSB of a+b
            carry = MSB of a+b
```

FIGURE 2-3: Half Adder, designed to add 2 bits.

Full Adder: Now that we know how to add 2 bits, we present a *full-adder* chip, designed to add 3 bits. Like the half-adder case, the full-adder chip produces two outputs: the least significant bit of the addition, and the carry bit.

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

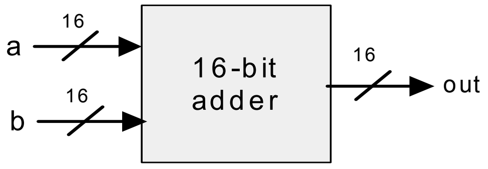


```
Chip name:  FullAdder
Inputs:     a, b, c
Outputs:    sum, carry
Function:   sum = LSB of a+b+c
            carry = MSB of a+b+c
```

FIGURE 2-4: Full Adder, designed to add 3 bits.

Adder: Memory and register chips represent integer numbers by *n*-bit patterns, *n* being 16, 32, 64, etc. – depending on the computer platform. The chip whose job is to add such numbers is called a multi-bit adder, or simply *adder*. We present a 16-bit adder, noting that an *n*-bit adder for any *n*-bit system is a simple extension.

…	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.
```

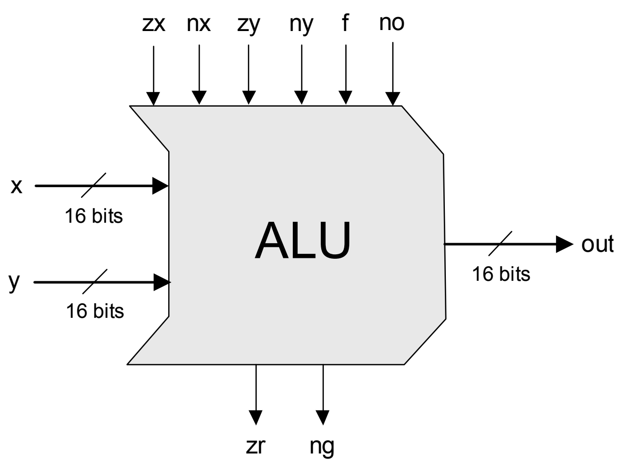
FIGURE 2-5: 16-bit adder. The example (top left) illustrates the addition of two 4-bit numbers. 16-bit addition is "more of the same."

Incrementer: It is convenient to have a special purpose chip dedicated to adding the constant 1 to a given number. Here is the API of a 16-bit incrementer:

```
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.
```

The Arithmetic Logic Unit (ALU)

This section presents a 16-bit ALU. This chip is designed to compute 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 possible functions. We instruct the ALU which function to compute by setting a set of six input bits, called *control bits*, to certain binary values. The exact specification of which function the ALU computes given each setting of the control bits is given in figure 2-6, using pseudo-code.



```
Chip name:  ALU
Inputs:     x[16],y[16],    // 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],        // data output
            zr,             // status flag, true when the ALU output=0
            ng              // status flag, true when the ALU output<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 equality comparison
            if out<0 then ng=1 else ng=0  // 2's-complement comparison
Comment:    overflow is neither detected nor handled.
```

FIGURE 2-6: The ALU interface diagram and API. The ALU operation (the function computed on x and y) is determined by the six control bits. The ALU sets the output bits zr and ng to 1 when the output out is zero or negative, respectively.

Note that each one of the six control bits instructs the ALU to carry out a certain operation. Taken together, the combined effects of these operations cause the ALU to compute a variety of useful functions. Since the ALU is controlled by six control bits, it can compute $2^6 = 64$ different functions. These include all the 18 useful functions documented in Table 2-7.

instructs how to pre-set the x input		instructs how to pre-set the y input		operation code	instructs how to post-set 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 And 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

TABLE 2-7: The ALU truth table (only 18 out of the 64 rows are shown). Taken together, the binary operations coded by the first six columns in each row affect the overall function listed in the right column of that row. (We use the symbols ~, &, and | to represent the operators Not, And, and Or, respectively, performed bit-wise.)

We see that programming the ALU to compute a certain function f(x,y) is done by setting the x and y inputs to the two 16-bit data values and selecting the function f by setting the six control bits to the code of the desired function. From this point on, the internal ALU logic specified in Figure 2-6 should cause the ALU to output the value f(x,y), as specified in Table 2-7.

Let us demonstrate this property on the 12th row of table 2-7, which instructs the ALU 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 bitwise. Bitwise negation of zero, 000...00, gives 111…11, which is the 2's complement code of -1. Thus the ALU is instructed to perform the operation on x and on the constant -1. Since the f bit is 1, the 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 Table 2-6 compute every one of the other 17 functions listed in the right column of Table 2-7? To verify that this is indeed the case, the reader is advised to pick up some other rows in the table and prove their respective ALU operation. We note in passing 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 table.

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 execute (right column in Table 2-7). 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, led to the design decision to use six control bits, each associated with a certain binary operation.

2.3 Implementation

As we have done in Chapter 1, our implementation guidelines are intentionally partial, since we want you to discover the actual chip architectures yourself. As usual, each gate can be implemented in more than one way; the simpler the implementation, the better.

Half Adder: An inspection of Figure 2-3 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) functions. Thus, the implementation of this adder is rather trivial, using previously built gates.

Full Adder: A Full-Adder chip can be implemented from two Half-Adder chips and a single simple gate. Other implementation options are 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 busses 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 3 bits. Hence, an *n*-bit adder can be implemented by creating an array of *n* Full-Adder chips, and chaining them in such a way that the carry bit of each adder is fed into one of the inputs of the next adder up the significance ladder.

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

ALU: Note that the ALU was carefully planned to effect all the desired ALU operations *logically*, using simple Boolean operations. Therefore, the *physical* implementation of the ALU is a matter of 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 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 addition and for bit-wise And-ing have already been built. Thus, what remains is to build logic that chooses between them according to the f control bit. Finally, you will need some logic that integrates all the other chips into the overall ALU.

2.4 Perspective

The construction of the multi-bit adder presented in this chapter was standard, although no attention was paid to efficiency considerations. In particular, our suggested adder implementation is rather inefficient, due to the long delays incurred while the carry propagates from the least significant bit to the most significant bit. 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 computer architecture, such low-level improvements 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 tradeoff 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 and division operations, nor floating point arithmetic. Some of these operations (as well as many more mathematical functions) will be implemented at the operating system level, as described in Chapter 11.

Detailed treatments of Boolean arithmetic and ALU design can be found in standard undergraduate textbooks such as [Hennessy & Patterson, chapter 4].

2.5 Build It

Objective: Implement all the chips presented in this chapter, using previously built chips.

Tip: When your HDL programs invoke chips from Chapter 1, it is recommended to use the built-in versions of these chips. This will ensure correctness and speed up the operation of the hardware simulator.

The remaining instructions for this project are identical to those of Project 1 (section 1.5), except that every occurrence of the text "project1" should be replaced with the text "project2".