2

The Language of Bits

A computer does not understand words or sentences like human beings. It understands only a sequence of 0s and 1s. We shall see in the rest of this book that it is very easy to store, retrieve and process billions of 0s and 1s. Secondly, existing technologies to implement computers using silicon transistors are very compatible with the notion of processing 0s and 1s. A basic silicon transistor is a *switch* that can set the output to a logical 0 or 1, based on the input. The silicon transistor is the basis of all the electronic computers that we have today right from processors in mobile phones to processors in supercomputers. Some early computers made in the late nineteenth century processed decimal numbers. They were mostly mechanical in nature. It looks like for the next few decades, students of computer architecture need to study the language of 0s and 1s in great detail.

Now, let us define some simple terms. A variable that can be either 0 or 1, is called a *bit*. A set of 8 bits is called a *byte*.

Definition 12
Bit: *A variable that can have two values: 0 or 1.*

Definition 13
Byte: *A sequence of 8 bits.*

In this chapter, we shall look at expressing different concepts in terms of bits. The first question is, " what can we do with our notion of bits?". Well it turns out that we can do everything that we could have done if our basic circuits were able to process normal decimal

numbers. We can divide the set of operations into two major types – logical and arithmetic. *Logical* operations express concepts of the form, "the red light is on AND the yellow light is on", or "the bank account is closed if the user is inactive AND the account is a current account." Arithmetic operations refer to operations such as addition, multiplication, subtraction, and division.

We shall first look at logical operations using bits in Section 2.1. Then, we shall look at methods to represent positive integers using 0s and 1s in Section 2.2. A representation of a number using 0s and 1s is also known as a *binary* representation. We shall then look at representing negative integers in Section 2.3, representing floating point numbers(numbers with a decimal point) in Section 2.4, and representing regular text in Section 2.5. Arithmetic operations using binary values will be explained in detail in Chapter 8.

Definition 14
Representation of numbers or text using a sequence of 0s and 1s is known as a binary *representation.*

2.1 Logical Operations

Binary variables (0 or 1) were first described by George Boole in 1854. He used such variables and their associated operations to describe logic in a mathematical sense. He defined a full algebra consisting of simple binary variables, along with a new set of operators, and basic operations. In the honor of George Boole, a binary variable is also known as a *Boolean* variable, and an algebraic system of Boolean variables is known as *Boolean algebra*.

Historical Note 1
George Boole(1815 – 1864) was a professor of mathematics at Queen's college, Cork, Ireland. He proposed his theory of logic in his book – An Investigation of the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities. During his lifetime, the importance of his work was not recognized. It was only in 1937 that Claude Shannon observed that it is possible to describe the behavior of electronic digital circuits using Boole's system.

Definition 15

Boolean variable *A variable that can take only two values – 0 or 1.*

Boolean algebra *An algebraic system consisting of Boolean variables and some special operators defined on them.*

2.1.1 Basic Operators

A simple Boolean variable can take two values – 0 or 1. It corresponds to two states of a system. For example, it can represent the fact that a light bulb is off(0) or on(1). It is easy to represent a Boolean variable in an electronic circuit. If the voltage on a wire is 0, then we are representing a logical 0. If the voltage is equal to the supply voltage V_{dd}, then we are representing a logical 1. We shall have an opportunity to read more about electronic circuits in Chapter 7.

Let us consider a simple Boolean variable, A. Let us assume that A represents the fact that a light bulb is on. If $A = 1$, then the bulb is on, else it is off. Then the logical complement or negation of A, represented by $\overline{A}$, represents the fact that the bulb is off. If $\overline{A} = 1$, then the bulb is off, otherwise, it is on. The logical complement is known as the NOT operator. Any Boolean operator can be represented by the means of a *truth table*, which lists the outputs of the operator for all possible combinations of inputs. The truth table for the NOT operator is shown in Table 2.1.

A	$\overline{A}$
0	1
1	0

Table 2.1: Truth table for the NOT operator

Let us now consider multiple Boolean variables. Let us consider the three bulbs in a typical traffic light – red, yellow, green. Let their states at a given time t be represented by the Boolean variables – R, Y, and G – respectively. At any point of time, we want one and only one of the lights to be on. Let us try to represent the first condition (one light on) symbolically using Boolean logic. We need to define the OR operator that represents the fact that either of the operands is equal to 1. For example, A OR B, is equal to 1, if $A = 1$ or $B = 1$. Two symbols for the OR operator are used in literature – '+' and '$\vee$'. In most cases '+' is preferred. The reader needs to be aware that '+' is not the same as the addition operator. The correct connotation for this operator needs to be inferred from the context. Whenever, there is a confusion, we will revert to the $\vee$ operator in this book. By default, we will use the '+' operator to represent Boolean OR. Thus, condition 1 is: $R + Y + G = 1$. The truth table for the OR operator is shown in Table 2.2.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.2: Truth table for the OR operator

Now, let us try to formalize condition 2. This states that only one light needs to be on. We can alternatively say that it is not possible to find a pair of bulbs that are on together. We

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.3: Truth table for the AND operator

need to define a new operator called the AND operator (represented by '.' or '$\wedge$'). A AND B is equal to 1, when both A and B are 1. The truth table for the AND operator is shown in Table 2.3. Now, $R.Y$ represents the fact that both the red and yellow bulbs are on. This is not possible. Considering all such pairs, we have condition 2 as: $R.Y + R.G + G.Y = 0$. This formula represents the fact that no two pairs of bulbs are on simultaneously.

We thus observe that it is possible to represent complex logical statements using a combination of Boolean variables and operators. We can say that NOT, AND, and OR, are basic operators. Let us derive a set of operators from them.

2.1.2 Derived Operators

Two simple operators namely NAND and NOR are very useful. NAND is the logical complement of AND (truth table in Table 2.4) and NOR is the logical complement of OR (truth table in Table 2.5).

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.4: Truth table for the NAND operator

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

Table 2.5: Truth table for the NOR operator

NAND and NOR are very important operators because they are known as *universal operators*. We can use just the NAND operator or just the NOR operator to construct any other operator. For more details the reader can refer to Kohavi and Jha [Kohavi and Jha, 2009].

Let us now define the XOR operator that stands for exclusive-or. A XOR B is equal to 1, when $A = 1, B = 0$, or $A = 0, B = 1$. The truth table is shown in Table 2.6. They symbol for XOR is $\oplus$. The reader can readily verify that $A \oplus B = A.\overline{B} + \overline{A}.B$ by constructing truth tables.

2.1.3 Boolean Algebra

Given Boolean variables and basic operators, let us define some rules of Boolean algebra.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.6: Truth table for the XOR operator

NOT Operator

Let us look at some rules governing the NOT operator.

1. **Definition:** $\overline{0} = 1$, and $\overline{1} = 0$ – This is the definition of the NOT operator.

2. **Double negation:** $\overline{\overline{A}} = A$ – The NOT of (NOT of A) is equal to A itself.

OR and AND Operators

1. **Identity:** $A + 0 = A$, and $A.1 = A$ – If we compute the OR of a Boolean variable, A, with 0, or AND with 1, the result is equal to A.

2. **Annulment:** $A + 1 = 1$, and $A.0 = 0$ – If we compute A OR 1, then the result is always equal to 1. Similarly, A AND 0, is always equal to 0 because the value of the second operand determines the final result.

3. **Idempotence:** $A + A = A$, and $A.A = A$ – The result of computing the OR or AND of A with itself, is A.

4. **Complementarity:** $A + \overline{A} = 1$, and $A.\overline{A} = 0$ – Either $A = 1$, or $\overline{A} = 1$. In either case $A + \overline{A}$ will have one term, which is 1, and thus the result is 1. Similarly, one of the terms in $A.\overline{A}$ is 0, and thus the result is 0.

5. **Commutativity:** $A.B = B.A$, and $A + B = B + A$ – The order of Boolean variables does not matter.

6. **Associativity:** $A + (B + C) = (A + B) + C$, and $A.(B.C) = (A.B).C$ – We are free to parenthesize expressions containing only OR or AND operators in any way we choose.

7. **Distributivity:** $A.(B + C) = A.B + A.C$, and $A + B.C = (A + B).(A + C)$ – We can use this law to open up a parenthesis and simplify expressions.

We can use these rules to manipulate expressions containing Boolean variables in a variety of ways. Let us now look at a basic set of theorems in Boolean algebra.

2.1.4 De Morgan's Laws

There are two De Morgan's laws that can be readily verified by constructing truth tables for the LHS and RHS.

$$\overline{A+B} = \overline{A}.\overline{B} \tag{2.1}$$

The NOT of $(A+B)$ is equal to the AND of the complements of A and B.

$$\overline{AB} = \overline{A} + \overline{B} \tag{2.2}$$

The NOT of $(A.B)$ is equal to the OR of the complements of A and B.

Example 5
Prove the consensus theorem: $X.Y + \overline{X}.Z + Y.Z = X.Y + \overline{X}.Z$.
Answer:

$$\begin{aligned} X.Y + \overline{X}.Z + Y.Z &= X.Y + \overline{X}.Z + (X+\overline{X}).Y.Z \\ &= X.Y + \overline{X}.Z + X.Y.Z + \overline{X}.Y.Z \\ &= X.Y.(1+Z) + \overline{X}.Z.(1+Y) \\ &= X.Y + \overline{X}.Z \end{aligned} \tag{2.3}$$

Example 6
Prove the theorem: $(X+Z).(\overline{X}+Y) = X.Y + \overline{X}.Z$.
Answer:

$$\begin{aligned} (X+Z).(\overline{X}+Y) &= X.\overline{X} + X.Y + Z.\overline{X} + Z.Y \\ &= 0 + X.Y + \overline{X}.Z + Y.Z \\ &= X.Y + \overline{X}.Z + Y.Z \\ &= X.Y + \overline{X}.Z \quad (\textit{see Example 5}) \end{aligned} \tag{2.4}$$

2.1.5 Logic Gates

Let us now try to implement circuits to realize complex Boolean formulae. We will discuss more about this in Chapter 7. We shall just provide a conceptual treatment in this section. Let us define the term "logic gate" as a device that implements a Boolean function. It can be constructed from silicon, vacuum tubes, or any other material.

Definition 16
A logic gate *is a device that implements a Boolean function.*

Given a set of logic gates, we can design a circuit to implement any Boolean function. The symbols for different logic gates are shown in Figure 2.1.

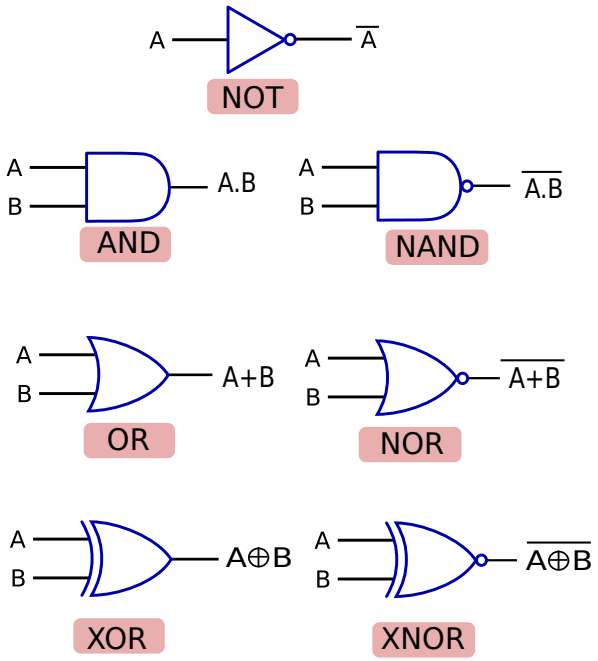


Figure 2.1: List of logic gates

2.1.6 Implementing Boolean Functions

Let us now consider a generic boolean function $f(A, B, C \ldots)$. To implement it we need to create a circuit out of logic gates. Our aim should be to minimize the number of gates to save area, power, and time. Let us first look at a brute force method of implementing any Boolean function.

Simple Method

We can construct the truth table of the function, and then try to realize it with an optimal number of logic gates. The reason we start from a truth table is as follows. In some cases, the Boolean function that we are trying to implement might not be specified in a concise form. It might be possible to simplify it significantly. Secondly, using truth tables ensures that the

process can be automated. For example, let us consider the following truth table of some function, f. We show only those lines that evaluate to 1.

A	B	C	Result
1	1	0	1
1	1	1	1
1	0	1	1

Let us consider the first line. It can be represented by the Boolean function $A.B.\overline{C}$. Similarly, the second and third lines can be represented as $A.B.C$ and $A.\overline{B}.C$ respectively. Thus, the function can be represented as:

$$f(A, B, C) = A.B.\overline{C} + A.B.C + A.\overline{B}.C \tag{2.5}$$

We see that we have represented the function as an OR function of several terms. This representation is known as a *sum-of-products* representation, or a representation in the *canonical form*. Each such term is known as a *minterm*. Note that in a minterm, each variable appears only once. It is either in its original form or in its complemented form.

Definition 17
Let us consider a Boolean function f with n arguments.

minterm *A* minterm *is an AND function on all n Boolean variables, where each variable appears only once (either in its original form or in its complemented form). A minterm corresponds to one line in the truth table, whose result is 1.*

Canonical representation *It is a Boolean formula, which is equivalent to the function f. It computes an OR operation of a set of minterms.*

To summarize, to implement a truth table, we first get a list of minterms that might evaluate to a logical 1 (*true*), then create a canonical representation, and then realize it with logic gates. To realize the canonical representation using logic gates, we need to realize each minterm separately, and then compute an OR operation.

This process works, but is inefficient. The formula: $A.B.\overline{C} + A.B.C + A.\overline{B}.C$, can be simplified as $A.B + A.\overline{B}.C$. Our simple approach is not powerful enough to simplify this formula.

Karnaugh Maps

Instead of directly converting the canonical representation into a circuit, let us build a structure called a *Karnaugh map*. This is a rectangular grid of cells, where each cell represents one minterm. To construct a Karnaugh map, let us first devise a method of numbering each minterm. Let us first represent all minterms such that the order of variables in them is the same (original or complemented). Second, if a variable is not complemented, then let us represent it

Minterm	Representation
$\overline{A}.\overline{B}.\overline{C}$	000
$\overline{A}.\overline{B}.C$	001
$\overline{A}.B.\overline{C}$	010
$A.\overline{B}.\overline{C}$	100
$\overline{A}.B.C$	011
$A.\overline{B}.C$	101
$A.B.\overline{C}$	110
$A.B.C$	111

Table 2.7: Representation of minterms

by 1, otherwise, let us represent it by 0. Table 2.7 shows the representation of all the possible 8 minterms in a three variable function.

Now, given the representation of a minterm we use some bits to specify the row in the Karnaugh map, and the rest of the bits to specify the column. We number the rows and columns such that adjacent rows or columns differ in the value of only one variable. We treat the last row, and the first row as adjacent, and likewise, treat the first and last columns as adjacent. This method of numbering ensures that the difference in representation across any two adjacent cells (same row, or same column) in the Karnaugh map is in only one bit. Moreover, this also means that the corresponding minterms differ in the value of only one variable. One minterm contains the variable in its original form, and the other contains it in its complemented form.

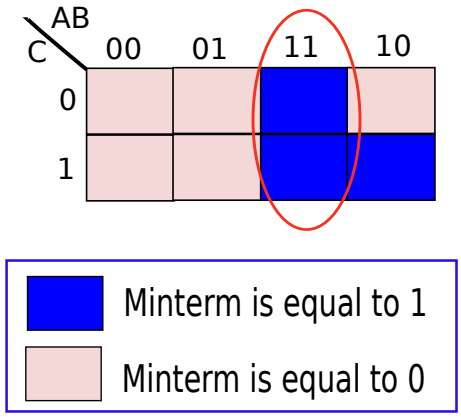


Figure 2.2: Karnaugh Map for $f(A,B,C) = A.B.\overline{C}(110) + A.B.C(111) + A.\overline{B}.C(101)$

Now, let us proceed to simplify or minimize the function. We construct the Karnaugh map as shown in Figure 2.2 for our simple function $f(A,B,C) = A.B.\overline{C} + A.B.C + A.\overline{B}.C$. We mark all the cells(minterms) that are 1 using a dark color. Let us consider the first minterm, $A.B.\overline{C}$. Its associated index is 110. We thus, locate the cell 110 in the Karnaugh map, and mark it. Similarly, we mark the cells for the other minterms – $A.B.C(111)$, and $A.\overline{B}.C(101)$.

We see that we have three marked cells. Furthermore, since adjacent cells differ in the value of only one variable, we can combine them to a single Boolean expression. In Figure 2.2, we try to combine the cells with indices 110, and 111. They differ in the value of the Boolean variable, C. After combining them, we have the boolean expression: $A.B.\overline{C} + A.B.C = A.B$. We have thus replaced two minterms by a smaller yet equivalent Boolean expression. We were able to combine the two adjacent cells, because they represented a logical OR of the Boolean expressions, which had the variable C in both its original and complemented form. Hence, the function f gets minimized to $A.B + A.\overline{B}.C$.

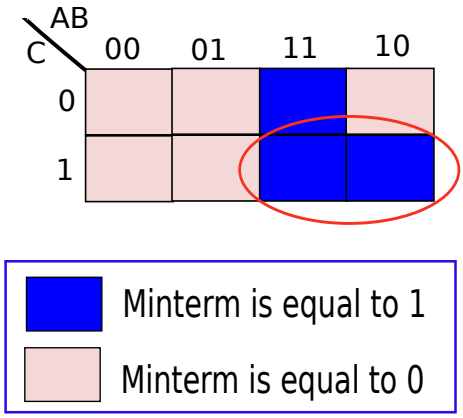


Figure 2.3: Karnaugh Map for $f(A, B, C) = A.B.\overline{C}(110) + A.B.C(111) + A.\overline{B}.C(101)$

Instead of combining, two cells in the same column, let us try to combine two cells in the same row as shown in Figure 2.3. In this case, we combine the minterms, $A.B.C$, and $A.\overline{B}.C$. Since the variable B is present in both its original and complemented forms, we can eliminate it. Thus, the Boolean expression denoting the combination of the cells is $A.C$. Hence, function f is equal to $A.C + A.B.\overline{C}$. We can readily verify that both the representations for f – $(A.C + A.B.\overline{C})$ and $(A.B + A.\overline{B}.C)$, are equivalent and optimal in terms of the number of Boolean terms.

Note that we cannot arbitrarily draw rectangles in the Karnaugh map. They cannot include any minterm that evaluates to 0 in the truth table. Secondly, the size of each rectangle needs to be a power of 2. This is because to remove n variables from a set of m minterms, we need to have all combinations of the n variables in the rectangle. It thus needs to have 2^n minterms.

To minimize a function, we need to draw rectangles that are as large as possible. It is possible that two rectangles might have an overlap. However, one rectangle should not be a strict subset of the other.

2.1.7 The Road Ahead

Way Point 2

- *Boolean algebra is a symbolic algebra that uses Boolean variables, which can be either 0 or 1.*
- *The basic Boolean operators are AND, OR, and NOT.*
- *These operators are associative, commutative, and reflexive.*
- *NAND, NOR, XOR are very useful Boolean operators.*
- *De Morgan's laws help convert an expression with an AND operator, to an expression that replaces it with an OR operator.*
- *A logic gate is a physical realization of a simple Boolean operator or function.*
- *Our aim is to minimize the number of logic gates while designing a circuit for a Boolean function.*
- *One effective way of minimizing the number of logic gates is by using Karnaugh maps.*

Up till now, we have learned about the basic properties of Boolean variables, and a simple method to design efficient circuits to realize Boolean functions. An extensive discussion on Boolean logic or optimal circuit synthesis is beyond the scope of this book. Interested readers can refer to seminal texts by Kohavi [Kohavi and Jha, 2009] and [Micheli, 1994].

Nevertheless, we are at a position to appreciate the nature of Boolean circuits. Up till now, we have not assigned a meaning to sets of bits. We shall now see that sequences of bits can represent integers, floating point numbers, and strings(pieces of text). Arithmetic operations on such sequences of bits are described in detail in Chapter 8.

2.2 Positive Integers

2.2.1 Ancient Number Systems

Ever since man developed higher intelligence, he has faced the need to count. For numbers from one to ten, human beings can use their fingers. For example, the little finger of the left hand can signify one, and the little finger of the right hand can signify ten. However, for counting numbers greater than ten, we need to figure out a way for representing numbers. In the ancient world, two number systems prevailed – the Roman numerals used in ancient Rome, and the Indian numerals used in the Indian subcontinent.

The Roman numerals used the characters – I, II ... X, for the numbers 1 ... 10 respectively. However, there were significant issues for representing numbers greater than ten. For example, to represent 50, 100, 500, and 1000, Romans used the symbols L, C, D, and M respectively. To represent a large number, the Romans represented it as a sequence of symbols. The number 204 can be represented as CCIV (C + C + IV = 100 + 100 + 4). Hence, to derive the real value of a number, we need to scan the number from left to right, and keep on adding the values. To make things further complicated, there is an additional rule that if a smaller number is preceded

by a larger value, then we need to subtract it from the total sum. Note that there is no notion of negative numbers, and zero in this number system. Furthermore, it is extremely difficult to represent large numbers, and perform simple operations such as addition and multiplication.

The ancient Indians used a number system that was significantly simpler, and fundamentally more powerful. The Arabs carried the number system to Europe sometime after seventh century AD, and thus this number system is popularly known as the *Arabic* number system. The magic tricks used by ancient Indian mathematicians are the number '0', and the place value system. The Indian mathematicians used a sequence of ten symbols including zero, as the basic alphabet for numbers. Figure 2.4 shows ten symbols obtained in the Bakhshali manuscript obtained in the north west frontier province of modern Pakistan (dated seventh century AD). Each such symbol is known as a 'digit'.

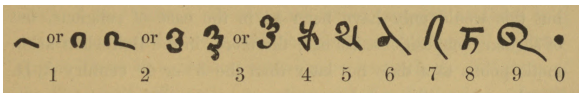


Figure 2.4: Numerals from the Bakhshali Manuscript (source Wikipedia®) This article uses material from the Wikipedia article "Bakhshali Manuscript" [bak,], which is released under the Creative Commons Attribution-Share-Alike License 3.0 [ccl,]

Every number was represented as a sequence of digits. Each digit represents a number between zero and nine. The first problem is to represent a number that is greater than nine by one unit. This is where we use the place value system. We represent it as 10. The left most number, 1, is said to be in the ten's place, and the right most number, 0, is in the unit's place. We can further generalize this representation to any two-digit number of the form, x_2x_1. The value of the number is equal to $10 \times x_2 + x_1$. As compared to the Roman system, this representation is far more compact, and can be extended to represent arbitrarily large integers.

A number of the form $x_n x_{n-1} \ldots x_1$ is equal to $x_n \times 10^{n-1} + x_{n-1} \times 10^{n-1} + \ldots + x_1 = \sum_{i=1}^{n} x_i 10^{i-1}$. Each decimal digit is multiplied with a power of 10, and the sum of the products is equal to the value of the number. As we have all studied in elementary school, this number system makes the job of addition, multiplication, and subtraction substantially easier. In this case, the number '10', is known as the *base* of the number system.

Historical Note 2
The largest number known to ancient Indian mathematicians was 10^{53} *[ind,].*

Let us now ponder about a basic point. Why did the Indians choose ten as the base? They had the liberty to choose any other number such as seven or eight or nine. The answer can be found by considering the most basic form of counting again, i.e., with fingers. Since human beings have ten fingers, they use them to count till one to ten, or from zero to nine. Hence, they were naturally inclined to use ten as the base.

Let us now move to a planet, where aliens have seven fingers. It would not be surprising to see them use a base seven number system. In their world, a number of the form, 56, would

actually be $7 \times 5 + 6$ in our number system. We thus observe that it is possible to generalize the concept of a *base*, and it is possible to represent any number in any base. We introduce the notation 3243_{10}, which means that the number 3243 is being represented in base 10.

Example 7
The number 1022_8 *is equal to :* $8^3 + 0 + 2 * 8^1 + 2 = 530_{10}$.

2.2.2 Binary Number System

What if we consider a special case? Let us try to represent numbers in base 2. The number 7_{10} can be represented as 111_2, and 12_{10} is equal to 1100_2. There is something interesting about this number system. Every digit is either 0 or 1. As we shall see in Chapters 7 and 8, computers are best suited to process values that are either 0 or 1. They find it difficult to process values from a larger set. Hence, representing numbers in base 2 should be a natural fit for computers. We call this a *binary number system* (see Definition 18). Likewise, a number system that uses a base of 10, is known as a *decimal* number system.

Definition 18

- *A number system based on Indian numerals that uses a base equal to 2, is known as a* **binary** *number system.*
- *A number system based on Indian numerals that uses a base equal to 10, is known as a* **decimal** *number system.*

Formally, any number A can be represented as a sequence of n binary digits:

$$A = \sum_{i=1}^{n} x_i 2^{i-1} \tag{2.6}$$

Here, $x_1 \ldots x_n$ are binary digits (0 or 1). We represent a number as a sum of the powers of 2, as shown in Equation 2.6. The coefficients of the equation, are the binary digits. For example, the decimal number 23 is equal to $(16 + 4 + 2 + 1) = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2 + 1$. Thus, its binary representation is 10111.

Let us consider some more examples, as shown in Table 2.8.

Number in decimal	Number in binary
5	101
100	1100100
500	111110100
1024	10000000000

Table 2.8: Examples of binary numbers

Example 8
Convert the decimal number 27 to binary.

Answer: $27 = \underbrace{16}_{1} + \underbrace{8}_{1} + \underbrace{0}_{0} + \underbrace{2}_{1} + \underbrace{1}_{1} = 11011_2$

Let us now define two more terms, the most significant bit (MSB), and the least significant bit (LSB). The LSB is the rightmost bit, and the MSB is the leftmost bit.

Definition 19

- *MSB (Most Significant Bit) : The leftmost bit of a binary number. For example the MSB of 1110 is 1.*
- *LSB (Least Significant Bit) : The rightmost bit of a binary number. For example the LSB of 1110 is 0.*

Hexadecimal and Octal Numbers

If we have a 32-bit number system, then representing each number in binary will take 32 binary digits (0/1). For the purposes of explanation, this representation is unwieldy. We can thus make our representation more elegant by representing numbers in base 8 or base 16. We shall see that there is a very easy method of converting numbers in base 8, or base 16, to base 2.

Numbers represented in base 8 are known as *octal numbers*. They are traditionally represented by adding a prefix, '0'. The more popular representation is the hexadecimal number system. It uses a base equal to 16. We shall use the hexadecimal representation extensively in this book. Numbers in this format are prefixed by '0x'. Secondly, the word 'hexadecimal' is popularly abbreviated as 'hex'. Note that we require 16 hex digits. We can use the digits 0-9 for the first ten digits. The next six digits require special characters. These six characters are typically – A (10), B(11), C(12), D(13), E(14), and F(15). We can use the lower case versions of ABCDEF also.

To convert a binary number (A) to a hexadecimal number, or do the reverse, we can use the following relationship:

$$\begin{aligned} A &= \sum_{i=1}^{n} x_i 2^{i-1} \\ &= \sum_{j=1}^{n/4} (2^3 \times x_{4(j-1)+4} + 2^2 \times x_{4(j-1)+3} + 2^1 \times x_{4(j-1)+2} + x_{4(j-1)+1}) \times 2^{4(j-1)} \\ &= \sum_{j=1}^{n/4} \underbrace{(2^3 \times x_{4(j-1)+4} + 2^2 \times x_{4(j-1)+3} + 2^1 \times x_{4(j-1)+2} + x_{4(j-1)+1})}_{y_j} \times 2^{4(j-1)} \\ &= \sum_{j=1}^{n/4} y_j 16^{(j-1)} \end{aligned} \tag{2.7}$$

We can thus represent the number (A) in base 16 (hexadecimal notation) by creating groups of four consecutive binary digits. The first group comprises the binary digits $x_4x_3x_2x_1$, the second group comprises $x_8x_7x_6x_5$, and so on. We need to convert each group of 4 binary digits, to represent a hexadecimal digit (y_j). Similarly, for converting a number from hex to binary, we need to replace each hex digit with a sequence of 4 binary digits. Likewise, for converting numbers from binary to octal and back, we need to consider sequences of 3 binary digits.

Example 9
Convert 110001010_2 *to the octal format.*
Answer: $\underbrace{110}_{6}\underbrace{001}_{1}\underbrace{010}_{2} \rightarrow$ *0612*

Example 10
Convert 110000101011 *to the hexadecimal format.*
Answer: $\underbrace{1100}_{C}\underbrace{0010}_{2}\underbrace{1011}_{B} \rightarrow$ *0xC2B*

2.2.3 Adding Binary Numbers

Adding binary numbers is as simple as adding decimal numbers. For adding decimal numbers, we start from the rightmost position and add digit by digit. If the sum exceeds 10, then we write the unit's digit at the respective position in the result, and carry the value at the ten's place to the next position in the result. We can do something exactly similar for binary numbers.

Let us start out by trying to add two 1-bit binary numbers, A and B. Table 2.9 shows the different combinations of numbers and results. We observe that for two bits, a carry is generated only when the input operands are both equal to 1. This carry bit needs to be added to the bits in the higher position. At that position, we need to add three bits – two input operand bits and a carry bit. This is shown in Figure 2.5. In this figure, the input operand bits are designated as A and B. The input carry bit is designated as C_{in}. The result will have two bits in it. The least significant bit (right most bit) is known as the *sum*, and the output carry is referred to as C_{out}.

Table 2.10 shows the results for the different combinations of input and carry bits.

A	B	$(A+B)_2$
0	0	00
0	1	01
1	0	01
1	1	11

Table 2.9: Addition of two binary bits

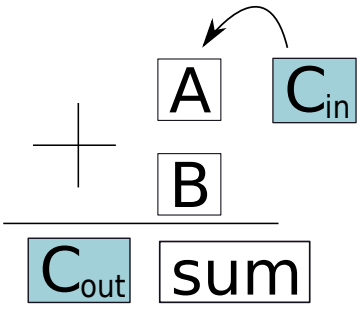


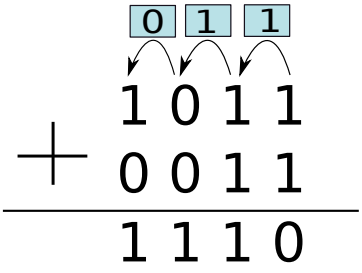
Figure 2.5: Addition of two binary bits and a carry bit

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Table 2.10: A truth table that represents the addition of three bits

Let us now try to add two n-bit binary numbers. Our addition needs to proceed exactly the same ways as decimal numbers. We add the values at a position, compute the result, and carry a value to the next (more significant) position. Let us explain with an example (see Example 11).

Example 11
Add the two binary numbers, 1011 and 0011.
Answer: *The process of addition is shown in the figure, and the values of the intermediate values of the carry bits are shown in shaded boxes. Let us now verify if the result of the addition is correct. The two numbers expressed in the decimal number system are 11 and 3. 11 + 3 = 14. The binary representation of 14 is 1110. Thus, the computation is correct.*



2.2.4 Sizes of Integers

Note that up till now we have only considered positive integers. We shall consider negative integers in Section 2.3. Such positive integers are known as *unsigned* integers in high level programming languages such as C or C++. Furthermore, high level languages define three types of unsigned integers – *short* (2 bytes), *int* (4 bytes), *long long int* (8 bytes). A *short unsigned* integer is represented using 16 bits. Hence, it can represent all the integers from 0 to $2^{16} - 1$ (for a proof, see Example 12). Likewise, a regular 32-bit unsigned integer can represent numbers from 0 till $2^{32} - 1$. The ranges of each data type are given in Table 2.11.

Example 12
Calculate the range of unsigned 2-byte short integers.

Answer: *A short integer is represented by 16 bits. The smallest short integer is represented by 16 zeros. It has a decimal value equal to 0. The largest short integer is represented by all 1s. Its value, V, is equal to $2^{15} + \ldots + 2^0 = 2^{16} - 1$. Hence, the range of unsigned short integers is 0 to $2^{16} - 1$.*

Example 13
Calculate the range of an n-bit integer.

Answer: 0 *to* $2^n - 1$.

Example 14
We need to represent a set of decimal numbers from 0 *till* $m - 1$. *What is the minimum number of binary bits that we require?*

Answer: *Let us assume that we use* n *binary bits. The range of numbers that we can represent is* 0 *to* $2^n - 1$. *We note that* $2^n - 1$ *needs to be at least as large as* m. *Thus, we have:*

$$\begin{aligned} & 2^n - 1 \geq m - 1 \\ & \Rightarrow 2^n \geq m \\ & \Rightarrow n \geq log_2(m) \\ & \Rightarrow n \geq \lceil log_2(m) \rceil \end{aligned}$$

Hence, the minimum number of bits that we require is $\lceil log_2(m) \rceil$.

Data Type	Size	Range
unsigned short int	2 bytes	0 to $2^{16} - 1$
unsigned int	4 bytes	0 to $2^{32} - 1$
unsigned long long int	8 bytes	0 to $2^{64} - 1$

Table 2.11: Ranges of unsigned integers in C/C++

Important Point 5
For the more mathematically inclined, we need to prove that for an n*-bit integer, there is a one to one mapping between the set of* n *bit binary numbers, and the decimal numbers,* 0 *to* $2^n - 1$. *In other words, every* n*-bit binary number has a unique decimal representation. We leave this as an exercise for the reader.*

2.3 Negative Integers

We represent a negative decimal number by adding a '-' sign before it. We can in principle do the same with a binary number, or devise a better representation.

Let us consider the generic problem first. For a number system consisting of a set of numbers, $\mathcal{S}$ (both positive and negative), we wish to create a mapping between each number in $\mathcal{S}$, and a sequence of zeros and ones. A sequence of zeros and ones can alternatively be represented as an unsigned integer. Thus, putting it formally, we propose to devise a method for representing both positive and negative integers as a function $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{N}$ that maps a set of numbers, $\mathcal{S}$, to a set of unsigned integers, $\mathcal{N}$. Let us define the function $SgnBit(u)$ of a number, u. It is equal to 1 when u is negative, and equal to 0 when u is positive or zero. Secondly, unless specified otherwise, we assume that all our numbers require n bits per storage in the next few subsections.

2.3.1 Sign-Magnitude based Representation

We can reserve a bit for representing the sign of a number. If it is equal to 0, then the number is positive, else it is negative. This is known as the *sign-magnitude* representation. Let us consider an n-bit integer. We can use the MSB as the designated signed bit, and use the rest of the number to represent the number's magnitude. The magnitude of a number is represented using $n-1$ bits. This is a simple and intuitive representation. In this representation, the range of the magnitude of an n-bit integer is from 0 till $2^{n-1}-1$. Hence, the number system has a range equal to $\pm(2^{n-1}-1)$. Note that there are two zeros – a positive zero($00\ldots0$) and a negative zero($10\ldots0$).

Formally, the mapping function – $\mathcal{F}(u)$ – where u is a number in the range of the number system, is shown in Equation 2.8.

$$\mathcal{F}(u) = SgnBit(u) \times 2^{n-1} + \mid u \mid \tag{2.8}$$

For example, if we consider a 4-bit number system, then we can represent the number -2 as 1010_2. Here, the MSB is 1 (represents a negative number), and the magnitude of the number is 010, which represents 2.

The issues with this system are that it is difficult to perform arithmetic operations such as addition, subtraction, and multiplication. For example in our 4-bit number system, -2 + 2, can be represented as 1010 + 0010. If we naively do simple unsigned addition, then the result is 1100, which is actually -6. This is the wrong result. We need to use a more difficult approach to add numbers.

2.3.2 The 1's Complement Approach

For positive numbers, let us use the same basic scheme that assigns the MSB to a dedicated sign bit, which is 0 in this case. Moreover, let the rest of the $(n-1)$ bits represent the number's magnitude. For a negative number, -u($u \geq 0$), let us simply flip all the bits of +u. If a bit is 0, we replace it by 1, and vice versa. Note that this operation flips the sign bit also, effectively negating the number. The number system can represent numbers between $\pm(2^{n-1}-1)$ like the sign-magnitude system.

Formally, the mapping function $\mathcal{F}$ is defined as:

$$\mathcal{F}(u) = \begin{cases} u & u \geq 0 \\ \sim (\mid u \mid) \; or \; (2^n - 1 - \mid u \mid) & u < 0 \end{cases} \tag{2.9}$$

Note that a bitwise complement($\sim$) is the same as subtracting the number from $11\ldots1$ ($2^n - 1$).

Let us consider some examples with a 4-bit number system. We represent the number 2 as 0010. Here the sign bit is 0, signifying that it is a positive number. To compute -2, we need to flip each bit. This process yields 1101. Note that the sign bit is 1 now.

The 1's complement approach also suffers from similar deficiencies as the sign magnitude scheme. First, there are two representations for zero. There is a positive zero - 0000, and a negative zero - 1111.

Second, adding two numbers is difficult. Let us try to add 2 and -2. 2 + (-2) = 0010 + 1101. Using simple binary addition, we get 1111, which is equal to 0(negative zero). Hence, in this case simple binary addition works. However, now let us try to add 1 to -0. We have: -0 + 1 = 1111 + 0001 = 0000. This leads to a mathematical contradiction. If we add one to zero, the result should be one. However, in this case, it is still zero! This means that we need to make the process of addition more sophisticated. This will slow down the process of addition and make it more complex.

2.3.3 Bias-based Approach

Let us adopt a different approach now. Let us assume that the unsigned representation of a number ($\mathcal{F}(u)$) is given by:

$$\mathcal{F}(u) = u + bias \tag{2.10}$$

Here, *bias* is a constant.

Let us consider several examples using a 4-bit number system. The range of unsigned numbers is from 0 to 15. Let the bias be equal to 7. Then, the actual range of the number system is from -7 to +8. Note that this method avoids some pitfalls of the sign-magnitude and 1's complement approach. First, there is only one representation for 0. In this case it is 0111. Second, it is possible to use standard unsigned binary addition to add two numbers with a small modification.

Let us try to add 2 and -2. 2 is represented as +9 or 1001_2. Likewise, -2, is represented as +5, or 0101_2. If we add 2 and -2, we are in effect adding the unsigned numbers 5 and 9. 5 + 9 = 14. This is not the right answer. The right answer should be 0, and it should be represented as 0111 or +7. Nonetheless, we can get the right answer by subtracting the bias, i.e., 7. 14 - 7 = 7. Hence, the algorithm for addition is to perform simple binary unsigned addition, and then subtract the bias. Performing simple binary subtraction is also easy (need to add the bias). Hence, in the case of addition, for two numbers, u and v, we have:

$$\mathcal{F}(u + v) = \mathcal{F}(u) + \mathcal{F}(v) - bias \tag{2.11}$$

However, performing binary multiplication is difficult. The bias values will create issues. In this case, if the real value of a number is A, we are representing it as $A + bias$. If we multiply A and B naively, we are in effect multiplying $A + bias$ and $B + bias$. To recover the correct result, AB, from $(A + bias) \times (B + bias)$ is difficult. We desire an even simpler representation.

2.3.4 The 2's Complement Method

Here are the lessons that we have learned from the sign-magnitude, 1's complement, and bias based approaches.

1. We need a representation that is simple.
2. We would ideally like to perform signed arithmetic operations, using the same kind of hardware that is used for unsigned numbers.
3. It is not desirable to have two representations for zero. The number zero, should have a single representation.

Keeping all of these requirements in mind, the 2's complement system was designed. To motivate this number system, let us consider a simple 4-bit number system, and represent the numbers in a circle. Let us first consider unsigned numbers. Figure 2.6 shows the numbers presented in a circular fashion. As we proceed clockwise, we increment the number, and as we proceed anti-clockwise, we decrement the number. This argument breaks at one point as shown in the figure. This is between 15 and 0. If we increment 15, we should get 16. However, because of the limited number of bits, we cannot represent 16. We can only capture its four low order bits which are 0000. This condition is also called an overflow. Likewise, we can also define the term, *underflow*, that means that a number is too small to be represented in a given number system (see Definition 20). In this book, we shall sometimes use the word "overflow" to denote both an overflow and an underflow. The reader needs to infer the proper connotation from the context.

Definition 20

overflow *An* overflow *occurs when a number is too large to be represented in a given number system.*

underflow *An* underflow *occurs when a number is too small to be represented in a given number system.*

Let us now take a look at these numbers slightly differently as shown in Figure 2.7. We consider the same circular order of numbers. However, after 7 we have -8 instead of +8. Henceforth, as we travel clockwise, we effectively increment the number. The only point of discontinuity is between 7 and -8. Let us call this point of discontinuity as the "break point". This number system is known as the 2's complement number system. We shall gradually refine the definition of a 2's complement number to make it more precise and generic.

Definition 21
The point of discontinuity in the number circle is called the break point.

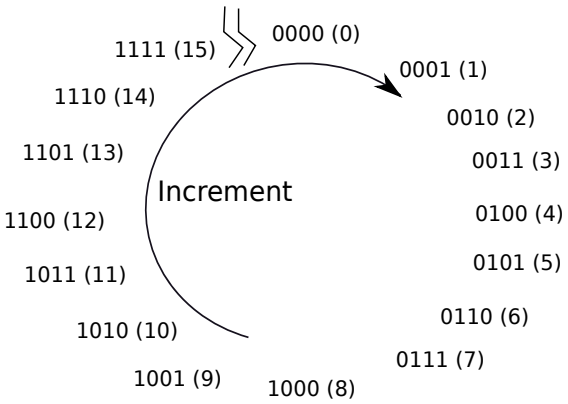


Figure 2.6: Unsigned 4-bit binary numbers

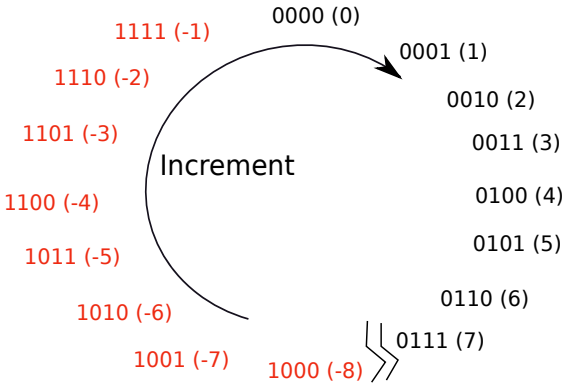


Figure 2.7: Signed 4-bit binary numbers

Let us now try to understand what we have achieved through this procedure. We have 16 numbers in the circle, and we have assigned each one of them to numbers from -8 to +7. Each number is represented by a 4-bit value. We observe that incrementing a signed number, is tantamount to incrementing its unsigned 4-bit representation. For example, -3 is represented as 1101. If we increment, -3, we get -2, which is represented as 1110. We also observe that 1101 + 1 = 1110.

Let us now try to formalize the pattern of numbers shown in the circle in Figure 2.7. First, let us try to give the circular representation a name. Let us call it a **Number Circle**. In a number circle, we observe that for numbers between 0 and 7, their representation is the same as their unsigned representation. The MSB is 0. For numbers between -8 and -1, the MSB is 1. Secondly, the representation of a negative number, -u ($u \geq 0$), is the same as the unsigned representation for $16 - u$.

Definition 22
The steps for creating an n-bit number circle are:

1. *We start by writing 0 at the top. Its representation is a sequence of n zeros.*
2. *We proceed clockwise and add the numbers 1 to ($2^{n-1}-1$). Each number is represented by its n-bit unsigned representation. The MSB is 0.*
3. *We introduce a break point after $2^{n-1} - 1$.*
4. *Then next number is -2^{n-1} represented by 1 followed by $n - 1$ zeros.*
5. *We then proceed clockwise incrementing both the numbers, and their unsigned representations by 1 till we reach 0.*

We can generalize the process of creating a number circle, to create an n-bit number circle (see Definition 22). To add a positive number, A, to a number B, we need to proceed A steps in the clockwise direction from B. If A is negative, then we need to proceed A steps in the anti-clockwise direction. Note that moving k steps in the clockwise direction is the same as moving $2^n - k$ steps in the anti-clockwise direction. This magical property means that subtracting k is the same as adding $2^n - k$. Consequently, every subtraction can be replaced by an addition. Secondly, a negative number, $-u$, is located in the number circle by moving $\mid u \mid$ steps anti-clockwise from 0, or alternatively, $2^n - \mid u \mid$ steps clockwise. Hence, the number circle assigns the unsigned representation $2^n - \mid u \mid$, to a negative number of the form $-u$ ($u \geq 0$).

Succinctly, a number circle can be described by Equation 2.12. This number system is called a *2's complement number system.*

$$\mathcal{F}(u) = \begin{cases} u & 0 \leq u \leq 2^{n-1} - 1 \\ 2^n - \mid u \mid & -2^{n-1} \leq u < 0 \end{cases} \tag{2.12}$$

Properties of the 2's Complement Representation

1. There is one unique representation for 0, i.e., $000\ldots0$.

2. The MSB is equal to the sign bit ($SgnBit(u)$).
   Proof: Refer to the number circle. A negative number's unsigned representation is greater than or equal to 2^{n-1}. Hence, its MSB is 1. Likewise, all positive numbers are less than 2^{n-1}. Hence, their MSB is 0.

3. **Negation Rule**: $\mathcal{F}(-u) = 2^n - \mathcal{F}(u)$
   Proof: If $u \geq 0$, then $\mathcal{F}(-u) = 2^n - u = 2^n - \mathcal{F}(u)$ according to Equation 2.12. Similarly, if $u < 0$, then $\mathcal{F}(-u) = |u| = 2^n - (2^n - |u|) = 2^n - \mathcal{F}(u)$.

4. Every number in the range $[-2^{n-1}, 2^{n-1} - 1]$ has a unique representation.
   Proof: Every number is a unique point on the number circle.

5. **Addition Rule**:

$$\mathcal{F}(u + v) \equiv \mathcal{F}(u) + \mathcal{F}(v) \tag{2.13}$$

   For the sake of brevity, we define the $\equiv$ operator. $(a \equiv b)$ means that $(a \mod 2^n = b \mod 2^n)$. Recall that the modulo (mod) operator computes the remainder of a division, and the remainder is assumed to be always non-negative, and less than the divisor. The physical significance of ($\mod 2^n$) is that we consider the n LSB bits. This is always the case because we have an n-bit number system, and in all our computations we only keep the n LSB bits, and discard the rest of the bits if there are any. In our number circle representation, if we add or subtract 2^n to any point (i.e. move 2^n hops clockwise or anti-clockwise), we arrive at the same point. Hence, $a \equiv b$ implies that they are the same point on the number circle, or their n LSB bits are the same in their binary representation.

   Proof: Let us consider the point u on the number circle. Its binary representation is $\mathcal{F}(u)$. Now, if we move v points, we arrive at $u + v$. If v is positive, we move v steps clockwise; otherwise, we move v steps anticlockwise. The binary representation of the new point is $\mathcal{F}(u + v)$.

   We can interpret the movement on the number circle in another way. We start at u, and move $\mathcal{F}(v)$ steps clockwise. If $v \geq 0$, then $v = \mathcal{F}(v)$ by Equation 2.12, hence we can conclude that we arrive at $u + v$. If $v < 0$, then $\mathcal{F}(v) = 2^n - |v|$. Now, moving $|v|$ steps anticlockwise is the same as moving $2^n - |v|$ steps clockwise. Hence, in this case also we arrive at $u+v$, which has a binary representation equal to $\mathcal{F}(u+v)$. Since, each step moved in a clockwise direction is equivalent to incrementing the binary representation by 1, we can conclude that the binary representation of the destination is equal to: $\mathcal{F}(u) + \mathcal{F}(v)$. Since, we only consider, the last n bits, the binary representation is equal to $(\mathcal{F}(u)+\mathcal{F}(v)) \mod 2^n$. Hence, $\mathcal{F}(u + v) \equiv \mathcal{F}(u) + \mathcal{F}(v)$.

6. **Subtraction Rule**

$$\mathcal{F}(u - v) \equiv \mathcal{F}(u) + (2^n - \mathcal{F}(v)) \tag{2.14}$$

   Proof: We have:

$$\begin{aligned}\mathcal{F}(u-v) &\equiv \mathcal{F}(u) + \mathcal{F}(-v) \quad \text{(addition rule)} \\ &\equiv \mathcal{F}(u) + 2^n - \mathcal{F}(v) \quad \text{(negation rule)}\end{aligned} \tag{2.15}$$

7. **Loop Rule**: $\mathcal{F}(u) \equiv 2^n + \mathcal{F}(u)$
   Proof: After moving 2^n points on the number circle, we come back to the same point.

8. **Multiplication Rule**: (assuming no overflows)

$$\mathcal{F}(u \times v) \equiv \mathcal{F}(u) \times \mathcal{F}(v) \tag{2.16}$$

   Proof: If u and v are positive, then this statement is trivially true. If u and v are negative, then we have, $u = -|u|$ and $v = -|v|$:

$$\begin{aligned}\mathcal{F}(u) \times \mathcal{F}(v) &\equiv (2^n - \mathcal{F}(|u|)) \times (2^n - \mathcal{F}(|v|)) \\ &\equiv 2^{n+1} - 2^n(\mathcal{F}(|u|) + \mathcal{F}(|v|) + \mathcal{F}(|u|) \times \mathcal{F}(|v|) \\ &\equiv \mathcal{F}(|u|) \times \mathcal{F}(|v|) \\ &\equiv \mathcal{F}(|u| \times |v|) \\ &\equiv \mathcal{F}(u \times v)\end{aligned} \tag{2.17}$$

   Now, let us assume that u is positive and v is negative. Thus, $u = |u|$ and $v = -|v|$. We have:

$$\begin{aligned}\mathcal{F}(u) \times \mathcal{F}(v) &\equiv \mathcal{F}(u) \times (2^n - \mathcal{F}(|v|)) \\ &\equiv 2^n\mathcal{F}(u) - \mathcal{F}(u) \times \mathcal{F}(|v|) \\ &\equiv -\mathcal{F}(u) \times \mathcal{F}(|v|) \quad (\textit{loop rule}) \\ &\equiv -(\mathcal{F}(u \times |v|)) \quad (u \geq 0, |v| \geq 0) \\ &\equiv 2^n - \mathcal{F}(u \times |v|) \quad (\textit{loop rule}) \\ &\equiv \mathcal{F}(-(u \times (|v|))) \quad (\textit{negation rule}) \\ &\equiv \mathcal{F}(u \times (-|v|)) \\ &\equiv \mathcal{F}(u \times v)\end{aligned} \tag{2.18}$$

   Likewise, we can prove the result for a negative u and positive v. We have thus covered all the cases.

We thus observe that the 2's complement number system, and the number circle based method make the process of representing both positive and negative numbers easy. It has a unique representation for zero. It is easy to compute its sign. We just need to take a look at the MSB. Secondly, addition, subtraction, and multiplication on signed numbers is as simple as performing the same operations on their unsigned representations.

Example 15
Add 4 and -3 using a 4-bit 2's complement representation.

Answer: *Let us first try to add it graphically. We can start at 4 and move 3 positions anti-clockwise. We arrive at 1, which is the correct answer. Let us now try a more conventional approach. 4 is represented as 0100, -3 is represented as 1101. If we add, 0100 and 1101 using a regular unsigned binary adder, the result is 10001. However, we cannot represent 5 bits in our simple 4-bit system. Hence, the hardware will discard the fifth bit, and report the result as 0001, which is the correct answer.*

Computing the 2's Complement Representation

Let us now try to explore the methods to compute a 2's complement representation. For positive numbers it is trivial. However, for negative numbers of the form, -u ($u \geq 0$), the representation is $2^n - u$. A simple procedure is outlined in Equation 2.19.

$$\begin{aligned} 2^n - u &= (2^n - 1 - u) + 1 \\ &= (\sim u) + 1 \end{aligned} \tag{2.19}$$

According to Equation 2.9, we can conclude that $(2^n - 1 - u)$ is equivalent to flipping every bit, or alternatively computing $\sim u$. Hence, the procedure for negating a number in the 2's complement system, is to first compute its 1's complement, and then add 1.

The Sign Extension Trick

Let us assume that we want to convert a number's representation from a 16-bit number system to a 32-bit number system. If the number is positive, then we just need to prefix it with 16 zeros. Let us consider the case when it is negative. Let the number again be of the form, -u ($u \geq 0$). Its representation in 16 bits is $\mathcal{F}_{16}(u) = 2^{16} - u$. Its representation using 32 bits is $\mathcal{F}_{32}(u) = 2^{32} - u$.

$$\begin{aligned} \mathcal{F}_{32}(u) &= 2^{32} - u \\ &= (2^{32} - 2^{16}) + (2^{16} - u) \\ &= \underbrace{11\ldots1}_{16}\underbrace{00\ldots0}_{16} + \mathcal{F}_{16}(u) \end{aligned} \tag{2.20}$$

For a negative number, we need to prefix it with 16 ones. By combining both the results, we conclude that to convert a number from a 16-bit representation to a 32-bit representation, we need to prefix it with 16 copies of its sign bit(MSB).

Range of the 2's Complement Number System

The range of the number system is from -2^{n-1} to $2^{n-1}-1$. There is one extra negative number, -2^{n-1}.

Checking if a 2's Complement Addition has Resulted in an Overflow

Let us outline the following theorem for checking if a 2's complement addition results in an overflow.

Theorem 2.3.4.1 *Let us consider an addition operation, where both the operands are non-zero. If the signs of the operands are different, then we can never have an overflow. However, if the signs of the operands are the same, and the result has an opposite sign or is zero, then the addition has led to an overflow.*

Proof: Let us consider the number circle, and an addition operation of the form $A + B$. Let us first locate point A. Then, let us move B steps clockwise if B is positive, or B steps anti-clockwise if B is negative. The final point is the answer. We also note that if we cross the break point (see Definition 21), then there is an overflow, because we exceed the range of the number system. Now, if the signs of A and B are different, then we need to move a minimum of $2^{n-1}+1$ steps to cross the break point. This is because we need to move over zero (1), the break point(1), and the set of all the positive numbers $(2^{n-1}-1)$, or all the negative numbers (2^{n-1}). Since, we have 1 less positive number, we need to move at least $2^{n-1}-1+1+1 = 2^{n-1}+1$ steps. Since B is a valid 2's complement number, and is in the range of the number system, we have $\mid B \mid < 2^{n-1}+1$. Hence, we can conclude that after moving B steps, we will never cross the break point, and thus an overflow is not possible.

Now, let us consider the case in which the operands have the same sign. In this case, if the result has an opposite sign or is zero, then we are sure that we have crossed the break point. Consequently, there is an overflow. It will never be the case that there is an overflow and the result has the same sign. For this to happen, we need to move at least $2^{n-1}+1$ steps (cross over 0, the break point, and all the positive/negative numbers). Like the earlier case, this is not possible.

Alternative Interpretation of 2's Complement

Theorem 2.3.4.2 *A signed n-bit number, A, is equal to ($A_{1\ldots n-1}$ - $A_n 2^{n-1}$). A_i is the i^{th} bit in A's 2's complement binary representation (A_1 is the LSB, and A_n is the MSB). $A_{1\ldots j}$ is a binary number containing the first j digits of A's binary 2's complement representation.*

Proof: Let us consider a 4-bit representation. -2 is represented as 1110_2. The last $n-1$ digits are 110_2. This is equal to 6 in decimal. The MSB represents 1000_2 or 8. Indeed -2 = 6 - 8.

If $A > 0$, then $A_n = 0$, and the statement of the theorem is trivially true. Let us consider the case when $A < 0$. Here, $A_n = 1$. We observe that $A_{1...n} = 2^n - |A| = 2^n + A$ since A is negative. Thus, $A = A_{1...n} - 2^n$.

$$\begin{aligned} A &= A_{1...n} - 2^n \\ &= (A_{1...n-1} + A_n 2^{n-1}) - 2^n \\ &= (A_{1...n-1} + 2^{n-1}) - 2^n \quad (A_n = 1) \\ &= A_{1...n-1} - 2^{n-1} \end{aligned} \tag{2.21}$$

■

2.4 Floating Point Numbers

Floating Point Numbers are numbers that contain a decimal point. Examples are: 3.923, -4.93, 10.23e-7 (10.23×10^{-7}). Note that the set of integers are a subset of the set of floating point numbers. An integer such as 7 can be represented as 7.0000000. We shall describe a method to represent floating point numbers in the binary format in this section.

In specific, we shall describe the IEEE 754 [Kahan, 1996] standard for representing floating point numbers. We shall further observe that representing different kinds of floating point numbers is slightly complicated, and requires us to consider many special cases. To make our life easy, let us first slightly simplify the problem and consider representing a set of numbers known as *fixed point numbers*.

2.4.1 Fixed Point Numbers

A fixed point number has a fixed number of digits after the decimal point. For example, any value representing money typically has two digits after the decimal point for most currencies in the world. In most cases, there is no reason for having more than three digits after the decimal point. Such numbers can be represented in binary easily.

Let us consider the case of values representing a monetary amount. These values will only be positive. A value such as 120.23 can be represented in binary as the binary representation of 12023. Here, the implicit assumption is that there are two digits after the decimal point. It is easy to add two numbers using this notation. It is also easy to subtract two numbers as long as the result is positive. However, multiplying or dividing such numbers is difficult.

2.4.2 Generic Form of Floating Point Numbers

Unlike fixed point numbers, there can potentially be many more digits after the decimal point in floating point numbers. We need a more generic representation. Let us first look at how we represent floating point numbers in a regular base-10 number system. For simplicity, let us limit ourselves to positive floating point numbers in this section.

Representing Floating Point Numbers in Base-10

Examples of positive floating point numbers in base 10 are: 1.344, 10.329, and 2.338. Alternatively, a floating point number, A, can be expanded according to Equation 2.22.

$$A = \sum_{i=-n}^{n} x_i 10^i \tag{2.22}$$

For example, $1.344 = 1 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} + 4 \times 10^{-3}$. The coefficient x_i can vary from 0 to 9. Let us try to use the basic idea in this equation to create a similar representation for floating point numbers in base 2.

Representing Floating Point Numbers in Binary

Let us try to extend the expansion shown in Equation 2.22 to expand positive floating point numbers in base 2. A is a positive floating point number. We can try to expand A as:

$$A = \sum_{i=-n}^{n} x_i 2^i \tag{2.23}$$

Here, x_i is either 0 or 1. Note that the form of Equation 2.23 is exactly the same as Equation 2.22. However, we have changed the base from 10 to 2.

We have negative exponents from -1 to $-n$, and non-negative exponents from 0 to n. The negative exponents represent the **fractional part** of the number, and the non-negative exponents represent the **integer part** of the number. Let us show a set of examples in Table 2.12. We show only non-zero co-coefficients for the sake of brevity.

Number	Expansion
0.375	$2^{-2} + 2^{-3}$
1	2^0
1.5	$2^0 + 2^{-1}$
2.75	$2^1 + 2^{-1} + 2^{-2}$
17.625	$2^4 + 2^0 + 2^{-1} + 2^{-3}$

Table 2.12: Representation of floating point numbers

We observe that using Equation 2.23, we can represent a lot of floating point numbers exactly. However, there are a lot of numbers such as 1.11, which will potentially require an infinite number of terms with negative exponents. It is not possible to find an exact representation for it using Equation 2.23. However, if n is large enough, we can reduce the error between the actual number and the represented number to a large extent.

Let us now try to represent a positive floating point number in a binary format using Equation 2.23. There are two parts in a positive floating point number – integer part and fractional part. We represent the integer part using a standard binary representation. We represent the fractional part also with a binary representation of the form: $x_{-1}x_{-2}\ldots x_{-n}$. Lastly, we put a '.' between the integer and fractional parts.

Number	Expansion	Binary Representation
0.375	$2^{-2} + 2^{-3}$	0.011
1	2^0	1.0
1.5	$2^0 + 2^{-1}$	1.1
2.75	$2^1 + 2^{-1} + 2^{-2}$	10.11
17.625	$2^4 + 2^0 + 2^{-1} + 2^{-3}$	10001.101

Table 2.13: Representation of floating point numbers in binary

Table 2.13 shows the binary representation of numbers originally shown in Table 2.12.

Normal Form

Let us take a look at Table 2.13 again. We observe that there are a variable number of binary bits before and after the decimal point. We can limit the number of bits before and after the decimal point to L_i and L_f respectively. By doing so, we can have a binary representation for a floating point number that requires $L_i + L_f$ bits – L_i bits for the integer part, and L_f bits for the fractional part. The fractional part is traditionally known as the *mantissa*, whereas the entire number (both integer and fraction) is known as the *significand*. If we wish to devote 32 bits for representing a floating point number, then the largest number that we can represent is approximately $2^{16} = 65,536$ (if $L_i = L_f$), which is actually a very small number for most practical purposes. We cannot represent large numbers such as 2^{50}.

Let us thus, slightly modify our generic form to expand the range of numbers that we can represent. We start out by observing that 101110 in binary can be represented as 1.01110×2^5. The number 1.01110 is the *significand*. As a convention, we can assume that the first binary digit in the significand is 1, and the decimal point is right after it. Using this notation, we can represent all floating point numbers as:

$$A = P \times 2^X, \quad (P = 1 + M, 0 \leq M < 1, X \in \mathbf{Z}) \tag{2.24}$$

Definition 23

Significand *It is the part of the floating point number that just contains its digits. The decimal point is somewhere within the significand. The significand of* 1.3829×10^3 *is 1.3829.*

Mantissa *It represents the fractional part of the significand. The mantissa of* 1.3829×10^3 *is 0.3829.*

$\mathbf{Z}$ is the set of integers, P is the significand, M is the mantissa, and X is known as the exponent. This representation is slightly more flexible. It allows us to specify large exponents,

both positive and negative. Lastly, let us try to create a generic form for both positive and floating point numbers by introducing a sign bit, S. We show the resulting form in Equation 2.25 and refer to it as the *normal form* henceforth.

$$A = (-1)^S \times P \times 2^X, \quad (P = 1 + M, 0 \leq M < 1, X \in \mathbf{Z}) \tag{2.25}$$

If $S = 0$, the number is positive. If $S = 1$, the number is negative.

2.4.3 IEEE 754 Format for Representing Floating Point Numbers

Let us now try to represent a floating point number using a sequence of 32 bits. We shall describe the IEEE 754 format, which is the de facto standard for representing floating point numbers in binary.

Let us start with the normal form as shown in Equation 2.25. We observe that there are three variables in the equation: S(sign bit), M(mantissa), and X(exponent). Since all significands have 1 as their first digit, there is no need to explicitly represent it. We can assume that we have a 1 by default as the MSB of the significand, and we need to just represent the L_f bits of the mantissa. Secondly, since we are representing all our numbers in binary, the base is 2, and this can be assumed to be the default value. The IEEE 754 format thus proposes to apportion 32 bits as shown in Figure 2.8.

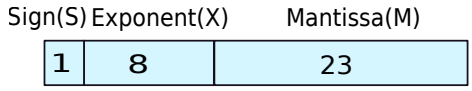


Figure 2.8: IEEE 754 format for representing 32-bit floating point numbers

The format allocates 1 bit for the sign bit, 8 bits for the exponent, and 23 bits for the mantissa. The exponent can be positive, negative or zero. The point to note here is that the exponent is not represented in the 2's complement notation. It is represented using the biased representation (see Section 2.3.3). The exponent(X) is represented by a number, E, where:

$$E = X + bias \tag{2.26}$$

In this case, the *bias* is equal to 127. Thus, if the exponent is equal 10, it is represented as 137. If the exponent is -20, it is represented as 107. E is an unsigned number between 0 and 255. 0 and 255 are reserved for special values. The valid range for E for **normal** floating point numbers is 1 to 254. Thus, the exponent can vary from -126 to 127. We can represent the normal form for IEEE 754 numbers as:

$$A = (-1)^S \times P \times 2^{E-bias}, \quad (P = 1 + M, 0 \leq M < 1, 1 \leq E \leq 254) \tag{2.27}$$

Example 16
Find the smallest and largest positive normal floating point numbers.
Answer:

- *The largest positive normal floating point number is* $1.\underbrace{11\ldots1}_{23} \times 2^{127}$.

$$\begin{aligned} 1.\underbrace{11\ldots1}_{23} &= 1 + \sum_{i=-1}^{-23} 2^i \\ &= \sum_{i=0}^{-23} 2^i \\ &= 2^1 - 2^{-23} \\ &= 2 - 2^{-23} \end{aligned}$$

  The result is equal to $(2 - 2^{-23}) \times 2^{127} = 2^{128} - 2^{104}$.

- *The smallest positive normal floating point number is* $1.00\ldots0 \times 2^{-126} = 2^{-126}$.

Example 17
What is the range of normal floating point numbers.
Answer: $\pm(2^{128} - 2^{104})$.

Special Numbers

We reserved two values of E, 0 and 255, to represent special numbers.

E	M	Value
255	0	∞ if $S = 0$
255	0	$-\infty$ if $S = 1$
255	$\neq 0$	NAN (Not a number)
0	0	0
0	$\neq 0$	Denormal number

Table 2.14: Special floating point numbers

If (E=255), then we can represent two kinds of values: ∞ and NAN (Not a number). We need to further look at the mantissa(M). If ($M = 0$), then the number represents $\pm\infty$ depending on the sign bit. We can get ∞ as a result of trying to divide any non-zero number by 0, or as the result of other mathematical operations. The point to note is that the IEEE 754 format treats infinities separately.

If we divide 0/0 or try to compute $sin^{-1}(x)$ for $x > 1$, then the result is invalid. An invalid result is known as a *NAN*. Any mathematical operation involving a NAN has as its result a

NAN. Even $NAN - NAN = NAN$. If $M \neq 0$, then the represented number is a NAN. In this case the exact value of M is not relevant.

Now, let us take a look at the case, when $E = 0$. If M is also 0, then the number represented is 0. Note that there are two 0s in the IEEE 754 format – a positive zero and a negative zero. Ideally implementations of this format are supposed to treat both the zeros as the same. However, this can vary depending upon the processor vendor.

The last category of numbers is rather special. They are called **denormal numbers**. We shall discuss them separately in Section 2.4.4.

2.4.4 Denormal Numbers

We have seen in Example 16 that the smallest positive normal floating point number is 2^{-126}. Let us consider a simple piece of code.

```
f = 2^(-126);
g = f / 2;
if (g == 0)
        print ("error");
```

Sadly, this piece of code will compute g to be 0 as per our current understanding. The reason for this is that f is the smallest possible positive number that can be represented in our format. g can thus not be represented, and most processors will round g to 0. However, this leads to a mathematical fallacy. The IEEE 754 protocol designers thus tried to avoid situations like this by proposing the idea of denormal numbers. Denormal numbers have a slightly different form as given by Equation 2.28.

$$A = (-1)^S \times P \times 2^{-126}, \quad (P = 0 + M, 0 \leq M < 1) \tag{2.28}$$

Note the differences with Equation 2.25. The implicit value of 1 is not there anymore. Instead of $(P = 1 + M)$, we have $(P = 0 + M)$. Secondly, there is no room to specify any exponent. This is because E=0. The default exponent is -126. We can view denormal numbers as an extension of normal floating point numbers on both sides (smaller and larger). Refer to Figure 2.9.

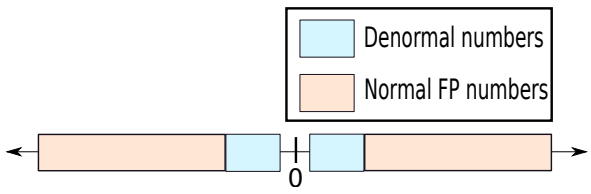


Figure 2.9: Denormal numbers on a conceptual number line (not drawn to scale)

Example 18
Find the smallest and largest positive denormal numbers.
Answer:

- *The smallest positive denormal number is* $0.\underbrace{00\ldots0}_{22}1 \times 2^{-126} = 2^{-149}$.
- *The largest possible denormal number is* $0.\underbrace{11\ldots1}_{23} \times 2^{-126} = 2^{-126} - 2^{-149}$.
- *Note that the largest denormal number* ($2^{-126} - 2^{-149}$) *is smaller than the smallest positive normal number* (2^{-126}). *This justifies the choice of* 2^{-126} *as the default exponent for denormal numbers.*

Example 19
Find the ranges of denormal numbers.
Answer:

- *For positive denormal numbers, the range is* $[2^{-149}, 2^{-126} - 2^{-149}]$.
- *For negative denormal numbers, the range is* $[-(2^{-126} - 2^{-149}), -2^{-149}]$.

By using denormal numbers we will not get a wrong answer if we try to divide 2^{-126} by 2, and then compare it with 0. Denormal numbers can thus be used as a buffer such that our normal arithmetic operations do not give unexpected results. In practice, incorporating denormal numbers in a floating point unit is difficult, and they require a lot of time to process. Consequently, a lot of small embedded processors do not support denormal numbers. However, most modern processors running on laptops and desktops have full support for denormal numbers.

2.4.5 Double Precision Numbers

We observe that by using 32 bits, the largest number that we can represent is roughly 2^{128}, which is approximately 10^{38}. We might need to represent larger numbers, especially while studying cosmology. Secondly, there are only 23 bits of precision (mantissa is 23 bits long). If we are doing highly sensitive calculations, then we might need more bits of precision. Consequently, there is an IEEE 754 standard for **double precision** numbers. These numbers require 64 bits of storage. They are represented by the *double* datatype in C or Java.

64 bits are apportioned as follows:

The mantissa is now 52 bits long. We have 11 bits for representing the exponent. The bias is equal to 1023, and the range of the exponent is from -1022 to 1023. We can thus represent many more numbers that are much larger, and we have more bits in the mantissa for added

Field	Size(bits)
S	1
E	11
M	52

precision. The format and semantics of $\pm\infty$, zero, NAN, and denormal numbers remains the same as the case for 32 bits.

2.4.6 Floating Point Mathematics

Because of limited precision, floating point formats do not represent most numbers accurately. This is because, we are artificially constraining ourselves to expressing a generic real number as a sum of powers of 2, and restricting the number of mantissa bits to 23. It is possible that some numbers such as 1/7 can be easily represented in one base (base 7), and can have inexact representations in other bases (base 2). Furthermore, there are a large set of numbers that cannot be exactly represented in any base. These are *irrational numbers* such as $\sqrt{2}$ or π. This is because a floating point representation is a rational number that is formed out of repeatedly adding fractions. It is a known fact that rational numbers cannot be used to represent numbers such as $\sqrt{2}$. Leaving theoretical details aside, if we have a large number of mantissa bits, then we can get arbitrarily close to the actual number. We need to be willing to sacrifice a tiny amount of accuracy for the ease of representation.

Floating point math has some interesting and unusual properties. Let us consider the mathematical expression involving two positive numbers A and B: $A + B - A$. We would ideally expect the answer to be non-zero. However, this need not be the case. Let us consider the following code snippet.

```
A = 2^(50);
B = 2^(10);
C = (B + A) - A;
```

Due to the limited number of mantissa bits (23), there is no way to represent $2^{50} + 2^{10}$. If the dominant term is 2^{50}, then our flexibility is only limited to numbers in the range $2^{50\pm23}$. Hence, a processor will compute $A+B$ equal to A, and thus C will be 0. However, if we slightly change the code snippet to look like:

```
A = 2^(50);
B = 2^(10);
C = B + (A - A);
```

C is computed correctly in this case. We thus observe that the order of floating point operations is very important. The programmer has to be either smart enough to figure out the right order, or we need a smart compiler to figure out the right order of operations for us. As we see, floating point operations are clearly not associative. The proper placement of brackets is crucial. However, floating point operations are commutative ($A + B = B + A$).

Due to the inexact nature of floating point mathematics, programmers and compilers need to pay special attention while dealing with very large or very small numbers. As we have also seen, if one expression contains both small and large numbers, then the proper placement of brackets is very important.

2.5 Strings

A string data type is a sequence of characters in a given language such as English. For example, "test", is a string of four characters. We need to derive a bitwise representation for it, the same way we devised a representation for integers. Traditionally, characters in the English language are represented using the ASCII character set. Hence, we shall describe it first.

2.5.1 ASCII Format

ASCII stands for "American Standard Code for Information Interchange". It is a format that assigns a 7-bit binary code for each English language character including punctuation marks. Most languages that use the ASCII format, use 8 bits to represent each character. One bit(MSB) is essentially wasted.

The ASCII character set defines 128 characters. The first 32 characters are reserved for control operations, especially for controlling the printer. For example, the zeroth character is known as the null character. It is commonly used to terminate strings in the C language. Similarly, there are special characters for backspace(8), line feed(10), and escape(27). The common English language characters start from 32 onwards. First, we have punctuation marks and special characters, then we have 26 capital letters, and finally 26 small letters. We show a list of ASCII characters along with their decimal encodings in Table 2.15.

Since ASCII can represent only 128 symbols, it is suitable only for English. However, we need an encoding for most of the languages in the world such as Arabic, Russian, French, Spanish, Swahili, Hindi, Chinese, Thai, and Vietnamese. The Unicode format was designed for this purpose. The most popular Unicode standard until recently was UTF-8.

2.5.2 UTF-8

UTF (Universal character set Transformation Format - 8 bit) can represent every character in the Unicode character set. The *Unicode* character set assigns an unsigned binary number to each character of most of the world's writing systems. UTF-8 encodes 1,112,064 characters defined in the Unicode character set. It uses 1-6 bytes for this purpose.

UTF-8 is compatible with ASCII. The first 128 characters in UTF-8 correspond to the ASCII characters. When using ASCII characters, UTF-8 requires just one byte. It has a leading 0. However, the first byte can contain extra information such as the total number of bytes. This is encoded by having leading ones followed by a zero in the first byte. For example, if the first byte is of the form 11100010, then it means that the character contains 3 bytes. Each continuation byte begins with 10. Most of the languages that use variants of the Roman script such as French, German, and Spanish require 2 bytes in UTF-8. Greek, Russian (Cyrillic), Hebrew, and Arabic, also require 2 bytes.

Character	Code	Character	Code	Character	Code
a	97	A	65	0	48
b	98	B	66	1	49
c	99	C	67	2	50
d	100	D	68	3	51
e	101	E	69	4	52
f	102	F	70	5	53
g	103	G	71	6	54
h	104	H	72	7	55
i	105	I	73	8	56
j	106	J	74	9	57
k	107	K	75	!	33
l	108	L	76	#	35
m	109	M	77	$	36
n	110	N	78	%	37
o	111	O	79	&	38
p	112	P	80	(	40
q	113	Q	81	)	41
r	114	R	82	*	42
s	115	S	83	+	43
t	116	T	84	,	44
u	117	U	85	.	46
v	118	V	86	;	59
w	119	W	87	=	61
x	120	X	88	?	63
y	121	Y	89	@	64
z	122	Z	90	^	94

Table 2.15: ASCII Character Set

UTF-8 is a standard for the World Wide Web. Most browsers, applications, and operating systems are required to support it. It is by far the most popular encoding as of 2012.

2.5.3 UTF-16 and UTF-32

UTF-8 has been superseded by UTF-16, and UTF-32. UTF-16 uses either 2 byte or 4 byte encodings to represents all the Unicode characters. It is primarily used by Java and the Windows operating system. UTF-32 encodes all characters using exactly 32 bits. It is rarely used since it is an inefficient encoding.

2.6 Summary and Further Reading

2.6.1 Summary

Summary 2

1. *In computer architecture, we represent information using the* language of bits. *A bit can either take the value of 0 or 1. A sequence of 8 bits is called a byte.*
2. *A variable representing a bit is also called a Boolean variable, and an algebra on such Boolean variables is known as Boolean algebra.*
3. (a) *The basic operators in Boolean algebra are logical OR, AND, and NOT.*
   (b) *Some derived operators are NAND, NOR, and XOR.*
   (c) *We typically use the De Morgan's laws (see Section 2.1.4) to simplify Boolean expressions.*
4. *Any Boolean expression can be represented in a canonical form as a logical OR of minterms. It can then be minimized using Karnaugh Maps.*
5. *We can represent positive integers in a binary representation by using a sequence of bits. In this case, we represent a number, A, as $x_n x_{n-1} \ldots x_1$, where $A = \sum_{i=1}^{n} x_i 2^{i-1}$.*
6. *The four methods to represent a negative integer are:*
   (a) *Sign Magnitude based Method*
   (b) *The 1's Complement Method*
   (c) *Bias based Method*
   (d) *The 2's Complement Method*
7. *The 2's complement method is the most common. Its main properties are as follows:*
   (a) *The representation of a positive integer is the same as its unsigned representation with a leading 0 bit.*
   (b) *The representation of a negative integer $(-u)$ is equal to $2^n - u$, in an n-bit number system.*
   (c) *To convert an m-bit 2's complement number to an n-bit 2's complement number, where $n > m$, we need to extend its sign by $n - m$ places.*
   (d) *We can quickly compute the 2's complement of a negative number of the form $-u$ $(u \geq 0)$, by computing the 1's complement of u (flip every bit), and then adding 1.*
   (e) *Addition, subtraction, and multiplication (ignoring overflows) of integers in the 2's complement representation can be done by assuming that the respective binary representations represent unsigned numbers.*

8. *Floating point numbers in the IEEE 754 format are always represented in their normal form.*
   (a) *A floating point number, A, is equal to*
   $$A = (-1)^S \times P \times 2^X$$
   S is the sign bit, P is the significand, and X is the exponent.
   (b) *We assume that the significand is of the form $1 + M$, where $0 \leq M < 1$. M is known as the mantissa.*
9. *The salient points of the IEEE 754 format are as follows:*
   (a) *The MSB is the sign bit.*
   (b) *We have an 8-bit exponent that is represented using the biased notation (bias equal to 127).*
   (c) *We do not represent the leading bit (equal to 1) in the significand. Furthermore, we represent the mantissa using 23 bits.*
   (d) *The exponents, 0 and 255, are reserved for special numbers – denormal numbers, NAN, zero, and $\pm\infty$.*
10. *Denormal numbers are a special class of floating point numbers, that have a slightly different normal form.*
    $$A = (-1)^S \times P \times 2^{-126}, (0 \leq P < 1, P = 0 + M)$$
11. *Floating point arithmetic is always approximate; hence, arithmetic operations can lead to mathematical contradictions.*
12. *We represent pieces of text as a contiguous sequence of characters. A character can either be encoded in the 7-bit ASCII format, or in the Unicode formats that use 1-4 bytes per character.*

2.6.2 Further Reading

Boolean algebra is a field of study by itself. Boolean formulae, logic, and operations form the basis of modern computer science. We touched upon some basic results in this chapter. The reader should refer to [Kohavi and Jha, 2009] for a detailed discussion on Boolean logic, Karnaugh Maps, and a host of other advanced techniques to minimize the number of terms in Boolean expressions. For Boolean logic and algebra, the reader can also consult [Gregg, 1998, Patt and Patel, 2003, Whitesitt, 2010] The next step for the reader is to read more about the synthesis and optimization of large digital circuits. The book by Giovanni de Michel [Micheli, 1994] can be a very helpful reference in this regard. Number systems such as 2's complement naturally lead to computer arithmetic where we perform complex operations on numbers. The

reader should consult the book by Zimmermann [Brent and Zimmermann, 2010]. For learning more about the representation of characters, and strings, especially in different languages, we refer the reader to the Unicode standard [uni,].

Exercises

Boolean Logic

Ex. 1 — A, B, C and D are Boolean variables. Prove the following results:

a) $A.B + \overline{A}.B + \overline{B}.C + \overline{B}.\overline{C} = 1$

b) $(\overline{A} + \overline{B}).(\overline{A} + B).(A + \overline{B}.D + C) = \overline{A}.\overline{B}.D + \overline{A}.C$

c) $\overline{\overline{A}.\overline{B} + \overline{B}.C} = A.\overline{C} + B$

d) $A.\overline{B} + \overline{A}.\overline{B} + A.\overline{B}.C.D = \overline{B}$

Ex. 2 — Construct a circuit to compute the following functions using only NOR gates.

a)$\overline{A}$

b)$A + B$

c)$A.B$

d)$A \oplus B$

Ex. 3 — Construct a circuit to compute the following functions using only NAND gates.

a)$\overline{A}$

b)$A + B$

c)$A.B$

d)$A \oplus B$

** **Ex. 4** — Prove that any Boolean function can be realized with just NAND or NOR gates. [HINT: Use the idea of decomposing a function into its set of minterms.]

Ex. 5 — Why are the first and last rows or columns considered to be adjacent in a Karnaugh Map?

Ex. 6 — Minimize the following Boolean functions using a Karnaugh Map.

a)$ABC + AB\overline{C} + \overline{A}BC$

b)$ABCD + A\overline{BC}D + A\overline{D}$

* **Ex. 7** — Consider the Karnaugh map of the function $A_1 \oplus A_2 \ldots \oplus A_n$. Prove that it looks like a chess board. Why cannot we minimize this expression further?

Integer Number Systems

Ex. 8 — Convert the following 8-bit binary numbers in 1's complement form to decimal.

a) 01111101

b) 10000000

c) 11111111

d) 00000000

e) 11110101

Ex. 9 — Convert the following unsigned numbers (in the given base) to decimal:

a) $(243)_5$

b) $(77)_8$

c) $(FFA)_{16}$

d) $(100)_4$

e) $(55)_6$

Ex. 10 — Do the following calculations on unsigned binary numbers and write the result as an unsigned binary number.

a) $1100110101 + 1111001101$

b) $110110110 + 10111001$

c) $11101110 - 111000$

d) $10000000 - 111$

Ex. 11 — What are the pros and cons of the 1's complement number system?

Ex. 12 — What are the pros and cons of the sign-magnitude number system?

Ex. 13 — What is a number circle? How is it related to the 2's complement number system?

Ex. 14 — What does the point of discontinuity on the number circle signify?

Ex. 15 — Why is moving k steps on the number circle in a clockwise direction equivalent to moving 2^n - k steps in an anti-clockwise direction? Assume that the number circle contains 2^n nodes.

Ex. 16 — What are the advantages of the 2's complement notation over other number systems?

Ex. 17 — Outline a method to quickly compute the 2's complement of a number.

Ex. 18 — Prove the following result in your own words:

$$\mathcal{F}(u - v) \equiv \mathcal{F}(u) + (2^n - \mathcal{F}(v)) \tag{2.29}$$

Ex. 19 — Let us define *sign contraction* to be the reverse of sign extension. What are the rules for converting a 32-bit number to a 16-bit number by using sign contraction? Can we do this conversion all the time without losing information?

Ex. 20 — What are the conditions for detecting an overflow while adding two 2's complement numbers?

Floating Point Number System

Ex. 21 — Describe the IEEE 754 format.

Ex. 22 — Why do we avoid representing the bit to the left of the decimal point in the significand?

Ex. 23 — Define denormal numbers. How do they help to extend the range of normal floating point numbers?

Ex. 24 — In the standard form of a denormal number, why is the exponent term equal to 2^{-126}? Why is it not equal to 2^{-127}?

Ex. 25 — Convert the following floating point numbers into the IEEE 32-bit 754 format. Write your answer in the hexadecimal format.

a) $-1 * (1.75 * 2^{-29} + 2^{-40} + 2^{-45})$

b) 52

Ex. 26 — What is the range of positive and negative denormal floating point numbers?

Ex. 27 — What will be the output of the following C code snippet assuming that the fractions are stored in an IEEE 32-bit 754 format:

```
float a=pow(2,-50);
float b=pow(2,-74);
float d=a;
for(i=0; i<100000; i++)
{
        d=d+b;
}
if(d>a)
        printf("%d",1);
else
        printf("%d",2);
```

Ex. 28 — We claim that the IEEE 754 format represents real numbers approximately. Is this statement correct?

* **Ex. 29** — Prove that it is not possible to exactly represent $\sqrt{2}$ even if we have an indefinitely large number of bits in the mantissa.

* **Ex. 30** — How does having denormal numbers make floating point mathematics slightly more intuitive?

* **Ex. 31** — What is the correct way for comparing two floating point numbers for equality?

** **Ex. 32** — Assume that the exponent e is constrained to lie in the range $0 \leq e \leq X$ with a bias of q , and the base is b . The significand is p digits in length. Use an IEEE 754 like encoding. However, you need to devote one digit to store the value to the left of the decimal point in the significand.

a) What are the largest and smallest positive values that can be written in normal form.

b) What are the largest and smallest positive values that can be written in denormal form.

* **Ex. 33** — Most of the floating point numbers cannot be represented accurately in hardware due to the loss of precision. However, if we choose some other representation, we can represent certain kinds of floating point numbers without error.

a) Give a representation for storing rational numbers accurately. Devise a normal form for it.

b) Can other floating point numbers such as $\sqrt{2}$ be represented similarly?

Ex. 34 — Design a floating point representation, for a base 3 system on the lines of the IEEE 754 format.

Strings

Ex. 35 — Convert the string "459801" to ASCII. The ASCII representation of 0 is 0x30. Assume that all the numbers are represented in the ASCII format in sequence.

Ex. 36 — Find the Unicode representation for characters in a non-English language, and compare it with the ASCII encoding.

Design Problems

Ex. 37 — In this section, we have minimized Boolean expressions using Karnaugh maps. We solved all our examples manually. This method is not scalable for expressions containing hundreds of variables. Study automated techniques for minimizing Boolean expressions such as the Quinn-McCluskey tabulation method. Write a program to implement this method.