1 Numbers and Computation

Hui Xu, xuh@fudan.edu.cn

Learning Objective:

- Understand the binary system for number representation.
- Understand how computations are carried out by the ALU.

1.1 Binary System

In digital systems, natural numbers are commonly represented using binary (base-2) numbers. Each binary digit, or *bit*, represents a power of 2. The position of each bit determines its weight, starting from the rightmost (least significant) bit. For example, the binary number 1011_2 is interpreted as:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11$$

The binary system is fundamental to digital computation, both for theoretical and physical reasons. At the heart of all digital systems lies a simple but powerful device: the *transistor*. We will discuss the topic in Section 1.2.

1.1.1 Natural Number

To represent a natural number in binary, we repeatedly divide it by 2 and record the remainders. For example, we can convert 11 to binary as follows:

$$11 \div 2 = 5 \quad \text{remainder } 1$$
$$5 \div 2 = 2 \quad \text{remainder } 1$$
$$2 \div 2 = 1 \quad \text{remainder } 0$$
$$1 \div 2 = 0 \quad \text{remainder } 1$$

Now we read the remainders backwards and obtain 1011. In this way, we can covert any natural number to binary with enough bits. Table 1.1 shows how the first few natural numbers are represented in binary, alongside their decimal (base-10) and hexadecimal (base-16) equivalents. Hexadecimal is often used as a compact, human-readable representation of binary values, especially in programming and debugging.

Table 1.1: Decimal, Binary and Hexadecimal Numbers (transposed)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Binary	0	01	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

1.1.2 Negative Integer

So far, we have focused on representing non-negative integers using binary. However, digital systems must also handle negative numbers. There are several candidate methods for representing signed integers in binary:

- *Sign-and-Magnitude:* The most significant bit represents the sign (0 for positive, 1 for negative), and the remaining bits represent the magnitude. For example, in 4-bit representation:

$$+3 = 0011, \quad -3 = 1011$$

  This method is conceptually simple but results in two representations of zero (e.g., 0000 and 1000).

- *One's Complement:* Negative numbers are represented by inverting all bits of the positive counterpart. Again, this leads to two representations of zero. in 4-bit one's complement:

$$+3 = 0011, \quad -3 = 1100.$$

  It also results in two representations of zero (e.g., 0000 and 1111).

- *Two's Complement:* The most widely used method in modern computers. To represent $-x$, take the binary of x, invert all bits, and add 1:

$$+3 = 0011, \quad -3 = 1101$$

Two' s complement is the standard representation for negative integers. It not only eliminates the ambiguity of zero but also allows the same arithmetic circuits to handle both positive and negative values seamlessly. In particular, subtraction can be performed using the same adder hardware as addition. For example, the 4-bit subtraction can be achieved as

$$5 - 3 = 5 + (-3) = 0101 + 1101 = \not{1}0010 = 2$$

Thinking Question

Why can subtraction be turned into addition in two's complement representation?

1.1.3 Floating-point Numbers

In many applications, especially those involving measurements, finance, or scientific computation, we need to represent numbers with fractional parts (*e.g.,* 3.14, 0.5). These are known as *decimals* or *real numbers*. In digital systems, there are two primary methods for representing such numbers in binary: *fixed-point* and *floating-point*.

Fixed-Point Representation: This method allocates a fixed number of bits for the integer part and the fractional part. For example, in an 8-bit fixed-point format with 4 bits for the integer part and 4 bits for the fraction:

$$0010.1100_2 = 2 + \frac{1}{2} + \frac{1}{4} = 2.75$$

How to determine the bits of the fraction part? This process is similar to determining the bits of the integer part. Instead of dividing by two, we multiply the fractional part by two iteratively until the fraction becomes zero or we reach the desired precision. Fixed-point representation is simple and efficient. However, it has a limited range and fixed precision.

Floating-Point Representation: This method provides a much larger range of values and varying degrees of precision. It represents a number as:

$$\text{value} = (-1)^s \times 1.m \times 2^e$$

Floating-point numbers are now widely used by computers. They generally follow the IEEE 754 standard, which defines two formats:

- *Single precision (32-bit):* 1 bit for sign, 8 bits for exponent, 23 bits for mantissa (fractional part).
- *Double precision (64-bit):* 1 bit for sign, 11 bits for exponent, 52 bits for mantissa.

Next, we demonstrate how to convert a decimal number into IEEE 754 single-precision format. We use the number 6.25 as an example. The first step is to convert the decimal number to binary. We represent the integer part 6 with 110, and the fractional part 0.25 with .01 which means 2^{-2}.

$$6.25 = 110.01_2$$

Then we shift the binary point so that there is exactly one digit to the left of the point.

$$110.01_2 = 1.1001 \times 2^2$$

Now, we can determine the sign bit is 0 because it is a positive number; the exponet is $2 + 127 = 129 = 10000001_2$ because the single-precision format uses an 8-bit exponent with bias 127; the fraction bits is $1001\,000\ldots0$ by dropping the leading 1 and filling remaining bits with 0. Therefore, the final floating-point number for 6.25 is

$$0\ 10000001\ 10010000000000000000000.$$

We can evaluate the value of the number as follows

$$(-1)^s \times 1.m \times 2^e = (-1)^0 \times 1.1001_2 \times 2^{129-127} = 6.25$$

Due to the nature of floating-point representation with varying precision, not all decimal numbers, even those with very simple fractional parts, can be exactly represented in binary. For example:

$$0.1_{10} = 0.0001100110011\ldots_2 \quad \text{(repeating)}$$

This leads to rounding errors in floating-point arithmetic, which can accumulate in sensitive computations. Understanding these limitations is essential when working with real numbers in software and hardware.

Thinking Question

What kinds of decimal numbers cannot be represented exactly in floating-point format?
Which integers cannot be represented exactly in IEEE-754 single-precision (f32) format?

1.1.4 More

Not only numbers, all information is represented as bits in computer systems. For example, the character "A" corresponds to the pattern 01000001 in the ASCII code, and entire documents are just long sequences of such patterns. Images and videos are also stored as bits: each pixel' s color and brightness

are described numerically, and the resulting arrays of bits form pictures and moving scenes. Sound too is digitized into samples, each sample represented by bits that record the amplitude of the waveform at a moment in time.

What distinguishes one type of information from another is not the bits themselves but the interpretation. A given pattern of 0s and 1s could be a number, a letter, a pixel color, or an instruction for the processor, depending on the context. Understanding that all data reduces to bits helps explain why computers can store, process, and transmit such a diverse range of content with the same underlying hardware.

1.2 Computation

This section explains how computations on bits are performed by the Central Processing Units (CPU). Overall, a CPU consists of several major components, including the Arithmetic Logic Unit (ALU). The ALU is constructed from logic gates that perform fundamental operations such as AND, OR, and NOT. These logic gates, in turn, are implemented using transistors. At the most basic level, transistors are fabricated from semiconductors, primarily silicon, which underpins the remarkable speed and compactness of modern processors.

1.2.1 Semiconductor

Pure silicon have limited electrical conductivity. To control and enhance conductivity, a process called doping is used, where small amounts of impurity atoms are added. There are two types:

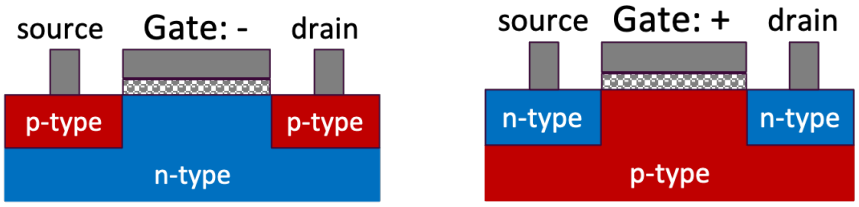
- *P-type semiconductors:* Atoms with three valence electrons, such as boron, are introduced, creating "holes" that act as positive charge carriers.
- *N-type semiconductors:* Atoms with five valence electrons, such as phosphorus, are added, supplying extra free electrons that serve as negative charge carriers.

By combining p-type and n-type materials, we can precisely control the flow of electric current in devices. An essential characteristic of the interface between the p-type and n-type regions, called the p–n junction, is that it allows current to flow easily in one direction (when the p-type side is connected to the positive terminal and the n-type side to the negative terminal) while blocking it in the opposite direction. This one-way conduction property is the basis for diodes and is fundamental to controlling current in transistors and other semiconductor devices.

1.2.2 Transistors

A transistor is an electronic switch that controls the flow of current in a circuit. It can turn current on or off, making it a fundamental building block of modern electronics. There are two common types of transistors used in digital circuits:

- *PMOS (P-type MOSFET):* It conducts when the gate voltage is low. As shown in Figure 1.1a, the channel of a PMOS transistor is full of holes (positive charge carriers). Applying a low voltage to the gate allows these holes to flow from the source to the drain, turning the transistor "on".
- *NMOS (N-type MOSFET):* It conducts when the gate voltage is high. Figure 1.1b demonstrates an example. The channel contains electrons (negative charge carriers). Applying a high voltage to the gate attracts electrons, creating a conductive path between the source and drain.



(a) PMOS (b) NMOS

Figure 1.1: Demonstration of PMOS and NMOS.

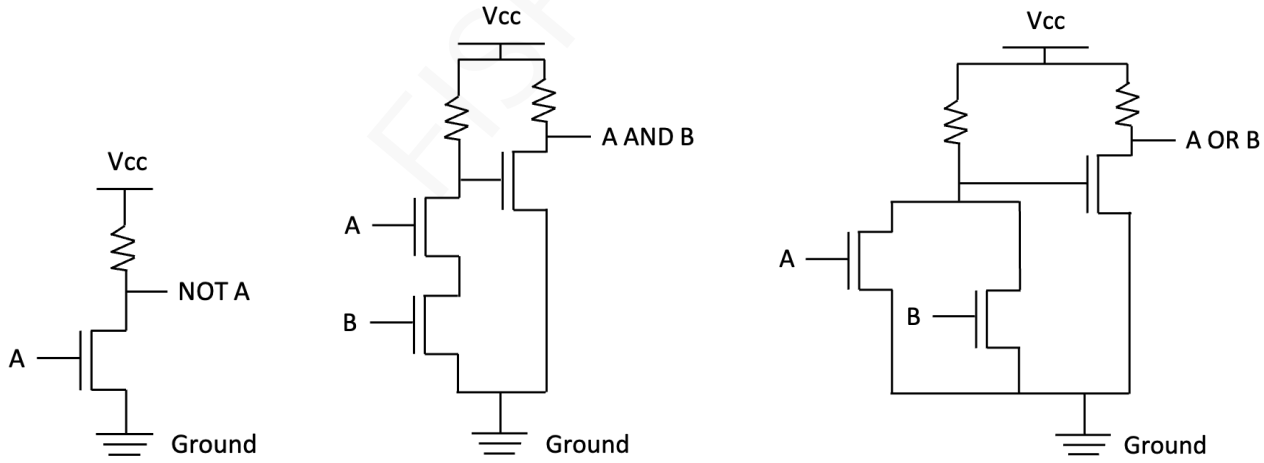
1.2.3 Logical Gates

With transistors, we can build circuits that perform logical operations such as `NOT`, `AND`, `OR`, and `XOR`. Table 1.2 shows the truth tables for these logical operations.

A	**B**	$\neg$**A**	**A** $\vee$ **B**	**A** $\wedge$ **B**	**A** $\oplus$ **B**
0	0	1	0	0	0
0	1	1	1	0	1
1	0	0	1	0	1
1	1	0	1	1	0

Table 1.2: Truth table for NOT ($\neg$), OR ($\vee$), AND ($\wedge$), and XOR ($\oplus$) gates.

Figure 1.3 shows three logic gates implemented with NMOS transistors. V_{CC} represents the supply voltage (the positive power rail) of the circuit, while ground is treated as 0 V. These circuits realize the corresponding truth table for each logic gate.



(a) NOT (b) AND (c) OR

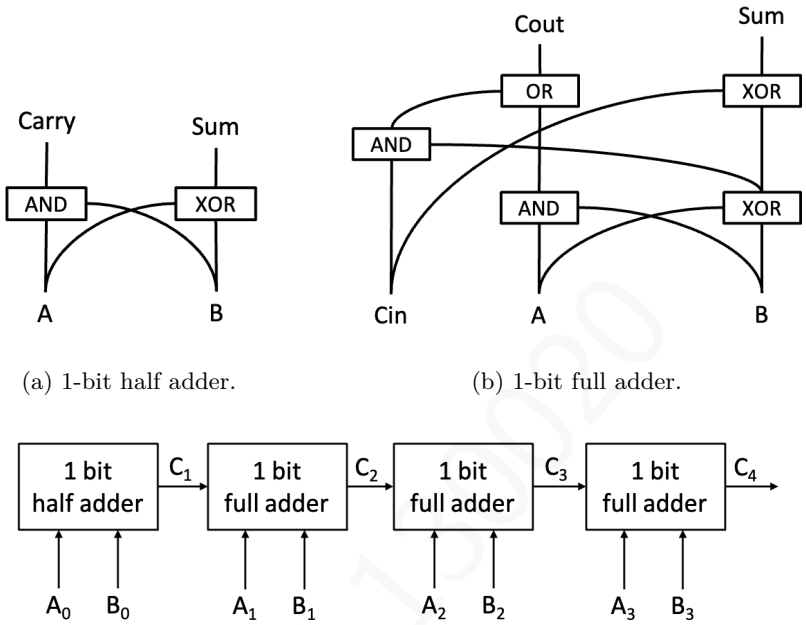
Figure 1.2: Logical gates composed with NMOS.

Taking the `NOT` gate in Figure 1.2a as an example, `A` is the input and `Not A` is the output. When no voltage is applied to `A` (low level), the NMOS transistor is off and `Not A` is pulled up to V_{CC} through the load, so the output is high. When a voltage is applied to `A` (high level), the NMOS turns on and connects `Not A` to ground, pulling the output low. The `AND` gate in Figure 1.2b is built using two NMOS transistors in series for the inputs `A` and `B`, and a load transistor connected to V_{CC}. When either `A` or `B` is low, the

series NMOS transistors are off, so no path to ground exists. The load transistor conducts, pulling the output high. Only when both `A` and `B` are high do the series NMOS transistors conduct, turning the load transistor off and pulling the output low. Similarly, the circuit in Figure 1.2c implements the `OR` logic function.

1.2.4 Arithmetic Logic Unit

ALUs are composed with logical gates. This section employs adder as an example to demonstrate the idea.



(a) 1-bit half adder.

(b) 1-bit full adder.

(c) Adder for 4-bit integers.

Figure 1.3: Composing adder with logical gates.

Figure 1.3a demonstrates a 1-bit half adder which is a simple digital circuit that adds two single-bit binary numbers, `A` and `B`. It produces two outputs: `Sum`, which is `A XOR B`, and `Carry`, which is the `A AND B`. The half adder' s truth table (Table 1.3) shows that when both inputs are 0, both outputs are 0; when exactly one input is 1, the Sum is 1 and Carry is 0; and when both inputs are 1, the Sum is 0 and Carry is 1. Since the half adder does not have a carry-in input, it can only be used for the least significant bit in multi-bit binary addition.

A	**B**	**Sum (A $\oplus$ B)**	**Carry (A $\cdot$ B)**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1.3: Truth table for 1-bit half adder.

Figure 1.3b presents a 1-bit full adder extends the half adder by including a carry-in from the previous stage. It has three inputs (`A`, `B`, and `Cin`) and two outputs (`Sum` and `Cout`). The Sum is computed as `A XOR B XOR Cin`, and `Cout` is `(A AND B) OR ((A XOR B) AND Cin)`. Table 1.4 shows the corresponding truth

table. The full adder can handle both the addition of two bits and any carry from the previous stage, making it suitable for constructing multi-bit adders by chaining multiple full adders together. Figure 1.3c demonstrates a simple adder for 4-bit integers.

A	**B**	**Cin**	**Sum (A $\oplus$ B $\oplus$ Cin)**	**Cout**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1.4: Truth table for 1-bit full adder.