

# Digital Computer Concept and Practice

Jaejin Lee

March 3, 2013



---

# Contents

---

<b>Contents</b>	<b>2</b>
<b>1 Introduction</b>	<b>5</b>
1.1 Computers . . . . .	5
1.2 The von Neumann Architecture . . . . .	6
1.3 The Stored Program Concept . . . . .	8
1.4 Files . . . . .	8
1.5 Operating Systems . . . . .	10
1.6 Programming Languages . . . . .	10
1.7 Abstractions in Computer Science . . . . .	12
<b>2 Number Systems</b>	<b>15</b>
2.1 Positional Number Systems . . . . .	15
2.2 Scientific Notation . . . . .	16
2.3 Binary Number System . . . . .	17
2.4 Number System Conversion . . . . .	19
2.5 Unsigned Binary Arithmetic . . . . .	22
2.6 Ones' Complement Representation . . . . .	23
2.7 Two's Complement Representation . . . . .	24
2.8 Shift Operations and Sign Extension . . . . .	25
2.9 Floating-Point Representation . . . . .	27
<b>3 Logic Circuits</b>	<b>29</b>
3.1 Boolean Algebra . . . . .	29
3.2 Logic Gates . . . . .	33
3.3 Combinational Logic Circuits . . . . .	36
3.4 Sequential Logic Circuits . . . . .	45
<b>4 A Simple Computer Architecture</b>	<b>59</b>
4.1 Datapath . . . . .	60
4.2 Attaching a RAM to the Datapath . . . . .	62
4.3 Register Transfer Language . . . . .	63

4.4	Programming the Datapath . . . . .	67
4.5	Instruction Set . . . . .	71
4.6	The Control Unit . . . . .	71
4.7	Assembly Language . . . . .	76
4.8	Input and Output . . . . .	79
	<b>Index</b>	<b>84</b>
	<b>Index</b>	<b>85</b>

# CHAPTER 1

---

## Introduction

---

### 1.1 Computers

A *computer* is an electronic device that accepts input, stores data, processes data according to a set of instructions (called program), and produces output in desired form. As shown in Figure 1.1, it can be abstracted as a black box that accepts input and produces output. The output depends on the current program in the block box.

Computer *input* is the information that is submitted to a computer by a human, by another computer, or by its environment. *Output* is the result produced by the computer, such as texts, audio, graphs, and pictures.

According to the Merriam-Webster Dictionary, data is factual information (as measurements or statistics) used as a basis for reasoning, discussion, or calculation[2]. In computer science, *data* is such factual information in a form suitable for use with a computer. Since a computer is an electronic device, the way to store data in the computer is in the form of electrical signals. These electrical signals typically have two states represented by 0 or 1. Thus, all data from outside a computer are transformed into a representation that uses 0 or 1 when stored or processed in the computer. The representation is transformed back to a desired form for output as a result of the computation.

A *bit* (a contraction of binary digit) is the fundamental unit of information in

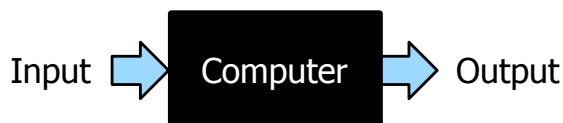


Figure 1.1: Computer as a black box.

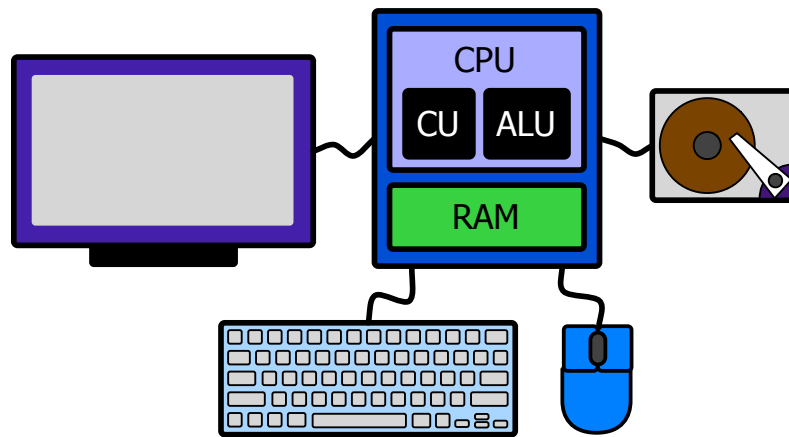


Figure 1.2: The von Neumann architecture.

computer science that has two possible distinct values, 0 or 1. Inside a computer, data is encoded as patterns of bits (i.e., 0s and 1s). A *bit pattern* is a sequence (or string) of bits. The meaning of a bit pattern depends on the interpretation. Sometimes it is used to represent numeric values; sometimes it represents other symbols such as characters in an alphabet; sometimes an image or audio. A *byte* is another unit of information. It consists of 8 bits. Historically, a byte was used to encode a single character of text.

While early computers were built for specific problems and solved mathematical and engineering problems, modern computers are aimed at more general problems in a variety of domains. Today, computers are even embedded in automobiles, airplanes, robots, refrigerators, microwave ovens, mobile phones, MP3 players, etc.

A computer system consists of hardware and software. *Hardware* is the physical components of the system. *Software* is the set of programs that instructs the hardware to obtain the output. A computer *program* is a set of instructions telling the computer what to do with the input in order to produce the output. It controls the actions of a machine (i.e., computer). The task of writing a program for a computer is called *programming*. The person who writes the program is called a *programmer*.

## 1.2 The von Neumann Architecture

Most of modern computers follow the *von Neumann architecture*. The term is derived from a proposal (*First Draft of a Report on the EDVAC*[3]) by the early computer scientist John von Neumann and others in 1945. They proposed the idea of a general-purpose electronic computer with a *stored program*. Von Neumann is not the first person who proposed the stored program concept for computers. There are other pioneers in computer science, such as Alan Turing,

J. Presper Eckert, and John Mauchly, who proposed the same concept. However, von Neumann receives the principal credit because he instructed the rest of the world about it.

A computer that follows the von Neumann architecture consists of four basic hardware components (Figure 1.2): input devices, output devices, main memory, and central processing unit (CPU).

There are many different types of *input devices*, including keyboards, mice, hard disk drives, bar code readers, etc. They transmit information from the outside world into main memory. For example, when you press a key on a computer keyboard, you send a character to the main memory of the computer. The character is encoded in a sequence of electrical signals. Then, the signals are transmitted to and stored in main memory.

*Output devices* transmit information from main memory to the outside world. They include screens, printers, hard disk drives, etc. For example, a character stored in main memory is transmitted to a monitor in a sequence of electrical signals. The monitor decodes the signals to display the character on its screen.

*Main memory* stores both the program and the data being processed. Data can be both read and written in main memory as bit patterns and the location of data does not affect the access speed. This type of memory is often referred to as *RAM* (random-access memory). In addition, main memory is typically *volatile*. That is, when the power is turned off, the information stored in main memory is lost. Main memory does not distinguish the type of data stored. Programs or input/output devices are responsible for interpreting the stored bit pattern as a number, a alphabetical symbol, or some other type.

*Machine code* is the representation of a program that is actually read, interpreted, and executed by the computer. A program in machine code consists of a sequence of *machine instructions*. A machine instruction is represented as a finite bit string.

The CPU carries out the instructions of a computer program. Two major components of a CPU are the *arithmetic logic unit* (ALU) and the *control unit* (CU). The ALU performs arithmetic and logical operations, and the CU fetches instructions from main memory, decodes them, and executes them. The CU uses the ALU to execute the instructions when necessary.

Main memory is treated as *primary storage*. Computers have *secondary storage* (alternatively referred to as *auxiliary storage* or *external memory*) that is non-volatile. It does not lose data stored when the power is down. It is typically used as both input and output devices, and for storing programs and data. The computer usually accesses its secondary storage through an intermediate space in main memory.

In modern computers, hard disk drives or solid-state disk drives are used as secondary storage. The capacity of secondary storage is typically two orders of magnitude bigger than that of main memory. Some other examples of secondary storage are flash memory (e.g., USB sticks), floppy disks, magnetic tape, punched cards, and paper tape.

### 1.3 The Stored Program Concept

Early computers were designed to perform a specific computing task. They had fixed programs. To use them for a different task, it was necessary to rewire, restructure, or redesign the hardware. This requires human intervention. For example, ENIAC (Electronic Numerical Integrator And Computer) constructed by the University of Pennsylvania in 1946 was programmed by setting switches and modifying wiring to route data and control signals.

Conceptually, programs and data are very different. The stored program concept means that we treat programs as data, and they can both be stored in main memory. With this, the program for a specific task is loaded into main memory (e.g., from secondary storage), and instructions in the program are executed one after another without any human intervention. The program is easily replaced by another program for a different task when necessary. By storing a program in main memory, the hours of tedious labor required to reprogram computers can be eliminated. A modern computer can solve almost an infinite variety of problems by just switching between different programs.

### 1.4 Files

According to the Merriam-Webster Dictionary, a *file* is a complete collection of data treated by a computer as a unit especially for purposes of input and output[2]. Computer files can be considered as the counterpart of traditional files that are kept in offices. The primary purpose of a computer file is to store data in a more permanent form. Files are typically stored in a secondary storage device.

The contents of a file are encoded in a sequence of bits. The meaning of the bits totally depends on the interpretation by the program that accesses the file. In general, there are two kinds of computer files: *text files* and *binary files*. The contents of a text file are interpreted as character symbols. Other files than text files are binary files. A binary file contains any type of data encoded in bits. Text files are considered to be different from binary files in general because binary files contain more than just textual data, such as formatting information encoded in bits.

A character is encoded as a bit string in a text file. *ASCII* (American Standard Code for Information Interchange) developed by ANSI (American National Standards Institute) is the most common character coding scheme for English-language text files. ASCII uses 7 bits for each character symbol. Thus, 128 ( $2^7$ ) different character symbols can be defined with ASCII. However, each byte in an ASCII text file contains a single character. Figure 1.3 shows the ASCII character table. ASCII was developed a long time ago and designed actually for use with teletypes. The first 32 non-printing characters are rarely used now.

Recently, the Unicode Consortium has developed a character coding scheme, called *Unicode*, to assign a unique value to every character symbol used in every language in the world. This allows texts from multiple languages to appear in



Dec	Bin	Char	Dec	Bin	Char	Dec	Bin	Char	Dec	Bin	Char
0	0000000	<b>NUL</b>	32	0100000	<b>SPC</b>	64	1000000	<b>@</b>	96	1100000	<b>`</b>
1	0000001	<b>SOH</b>	33	0100001	<b>!</b>	65	1000001	<b>A</b>	97	1100001	<b>a</b>
2	0000010	<b>STX</b>	34	0100010	<b>"</b>	66	1000010	<b>B</b>	98	1100010	<b>b</b>
3	0000011	<b>ETX</b>	35	0100011	<b>#</b>	67	1000011	<b>C</b>	99	1100011	<b>c</b>
4	0000100	<b>EOT</b>	36	0100100	<b>\$</b>	68	1000100	<b>D</b>	100	1100100	<b>d</b>
5	0000101	<b>ENQ</b>	37	0100101	<b>%</b>	69	1000101	<b>E</b>	101	1100101	<b>e</b>
6	0000110	<b>ACK</b>	38	0100110	<b>&amp;</b>	70	1000110	<b>F</b>	102	1100110	<b>f</b>
7	0000111	<b>BEL</b>	39	0100111	<b>'</b>	71	1000111	<b>G</b>	103	1100111	<b>g</b>
8	0001000	<b>BS</b>	40	0101000	<b>(</b>	72	1001000	<b>H</b>	104	1101000	<b>h</b>
9	0001001	<b>TAB</b>	41	0101001	<b>)</b>	73	1001001	<b>I</b>	105	1101001	<b>i</b>
10	0001010	<b>LF</b>	42	0101010	<b>*</b>	74	1001010	<b>J</b>	106	1101010	<b>j</b>
11	0001011	<b>VT</b>	43	0101011	<b>+</b>	75	1001011	<b>K</b>	107	1101011	<b>k</b>
12	0001100	<b>FF</b>	44	0101100	<b>,</b>	76	1001100	<b>L</b>	108	1101100	<b>l</b>
13	0001101	<b>CR</b>	45	0101101	<b>-</b>	77	1001101	<b>M</b>	109	1101101	<b>m</b>
14	0001110	<b>SO</b>	46	0101110	<b>.</b>	78	1001110	<b>N</b>	110	1101110	<b>n</b>
15	0001111	<b>SI</b>	47	0101111	<b>/</b>	79	1001111	<b>O</b>	111	1101111	<b>o</b>
16	0010000	<b>DLE</b>	48	0110000	<b>0</b>	80	1010000	<b>P</b>	112	1110000	<b>p</b>
17	0010001	<b>DC1</b>	49	0110001	<b>1</b>	81	1010001	<b>Q</b>	113	1110001	<b>q</b>
18	0010010	<b>DC2</b>	50	0110010	<b>2</b>	82	1010010	<b>R</b>	114	1110010	<b>r</b>
19	0010011	<b>DC3</b>	51	0110011	<b>3</b>	83	1010011	<b>S</b>	115	1110011	<b>s</b>
20	0010100	<b>DC4</b>	52	0110100	<b>4</b>	84	1010100	<b>T</b>	116	1110100	<b>t</b>
21	0010101	<b>NAK</b>	53	0110101	<b>5</b>	85	1010101	<b>U</b>	117	1110101	<b>u</b>
22	0010110	<b>SYN</b>	54	0110110	<b>6</b>	86	1010110	<b>V</b>	118	1110110	<b>v</b>
23	0010111	<b>ETB</b>	55	0110111	<b>7</b>	87	1010111	<b>W</b>	119	1110111	<b>w</b>
24	0011000	<b>CAN</b>	56	0111000	<b>8</b>	88	1011000	<b>X</b>	120	1111000	<b>x</b>
25	0011001	<b>EM</b>	57	0111001	<b>9</b>	89	1011001	<b>Y</b>	121	1111001	<b>y</b>
26	0011010	<b>SUB</b>	58	0111010	<b>:</b>	90	1011010	<b>Z</b>	122	1111010	<b>z</b>
27	0011011	<b>ESC</b>	59	0111011	<b>;</b>	91	1011011	<b>[</b>	123	1111011	<b>{</b>
28	0011100	<b>FS</b>	60	0111100	<b>&lt;</b>	92	1011100	<b>\</b>	124	1111100	<b> </b>
29	0011101	<b>GS</b>	61	0111101	<b>=</b>	93	1011101	<b>]</b>	125	1111101	<b>}</b>
30	0011110	<b>RS</b>	62	0111110	<b>&gt;</b>	94	1011110	<b>^</b>	126	1111110	<b>~</b>
31	0011111	<b>US</b>	63	0111111	<b>?</b>	95	1011111	<b>_</b>	127	1111111	<b>DEL</b>

Figure 1.3: The ASCII table.

a single text file. A character symbol in Unicode uses 16 bits (2 bytes). Thus, Unicode can represent up to 65,536 ( $2^{16}$ ) symbols. Different sections of Unicode are allocated to character symbols from different languages.

## 1.5 Operating Systems

*Application software* (also known as an *application*) is a set of programs that helps the user to carry out a specific task. For example, a *spreadsheet* is accounting software that helps the user to calculate numbers and organize information in a tabular form (e.g., columns and rows). In contrast, *system software* is a set of programs designed to operate the computer hardware and to provide a platform for running applications.

Especially, an *operating system* (also known as an *OS*) is system software that controls the operation of a computer and directs the processing of programs. It manages computer hardware resources and provides common services for application software. An application software asks a service to the operating system when necessary. The operating system assigns storage spaces in main memory, controls input and output functions, monitors the underlying computer system, etc. It is typical that a user cannot run an application on the computer without an operating system. Popular operating systems include Microsoft Windows, Mac OS, Linux, Unix, etc.

*Utility software* is system software designed to help the user manage and tune the computer hardware and software. It usually focuses on how the computer hardware and software operates. It is also referred to as *utility*, *tool*, and *service program*. Examples of utility software may include virus scanners, data compression utilities, disk partition utilities, archive utilities, system monitors, text editors, assemblers, etc.

## 1.6 Programming Languages

A *programming language* is a formal language in which computer programs are written. Most programmers write their programs in a *high-level programming language*, like Java, C, C++, FORTRAN, Scheme, ML, etc. The level of abstraction from the details of the underlying computer in a high-level programming language is higher than a *low-level programming language*. It is more close to natural languages and more understandable than a low-level language. Thus, a high-level programming language makes the process of developing programs simpler and easier.

An *assembly language* is a typical example of low-level programming languages. It represents machine instructions symbolically. The representation is defined by the hardware manufacturer and specific to a computer architecture. There exists an assembly instruction that corresponds to a machine instruction, but not *vice versa*. A program called an *assembler* is used to translate assembly language instructions into the target computer's machine code instructions.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6 }
```

Figure 1.4: An example C program hello.

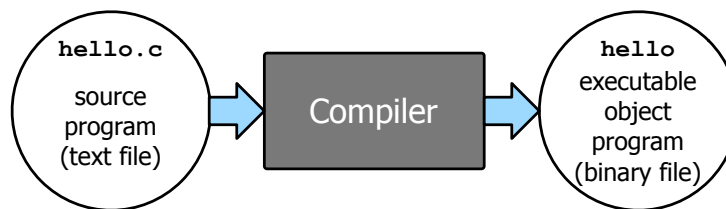


Figure 1.5: The compilation process.

The definition of a particular programming language consists of both *syntax* (how the various symbols of the language are combined) and *semantics* (the meaning of the language constructs). The syntax and semantics of a programming language are typically defined in its specification.

The C programming language was developed from 1969 to 1973 by Dennis Ritchie of Bell Laboratories. It was designed for a practical purpose to implement the UNIX operating system. As UNIX became popular, many software developers were exposed to C. Although it was originally designed as a systems programming language, it can be used for writing programs in a variety of different domains from business to engineering. It is one of the most widely used programming languages in these days.

In 1978, Brian Kernighan and Dennis Ritchie published the classical book on C, *The C Programming Language*[1]. This book was known to programmers as "K&R" and had served as an informal specification of C until 1989. In 1989, ANSI introduced C89 standard for the C programming language. The same standard was adopted by ISO (International Organization for Standardization) in 1990 and called C90. The standard was further revised, leading to ANSI/ISO C99 standard that was published in 1999. C99 is an internationally recognized C language specification, and almost all C compilers follow this standard. The standard promotes portability, reliability, maintainability, and efficient execution of C programs on a variety of machines.

Figure 1.4 shows an example C program hello. It is introduced in K&R, and prints hello, world on the screen when executed. The programmer creates the program with a *text editor* and saves it in a text file hello.c. The text file is stored in secondary storage. A *filename extension* c is used to indicate that the

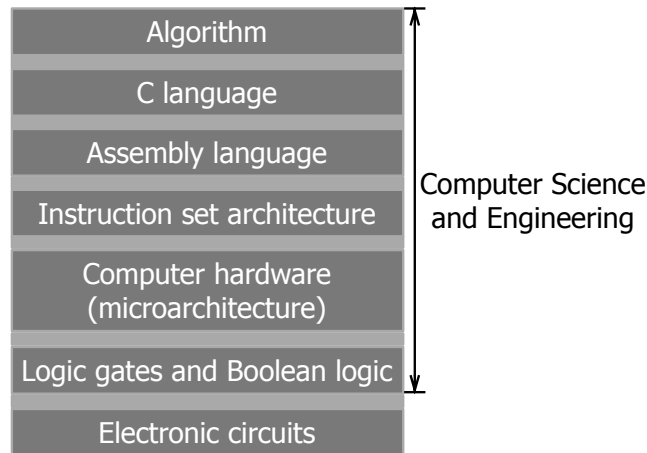


Figure 1.6: Abstractions in computer science and engineering.

text file contains a C program. A filename extension is a suffix to the filename (e.g., `c` that is separated from the base filename `hello` by a dot). It indicates the content type of the file. A text editor is a utility program to create and modify text files. Commonly used text editors include GNU emacs, UNIX vi, Microsoft word, etc.

To run the `hello` program on a computer, the C statements in the source program must be translated by a *compiler* into a sequence of machine instructions. A compiler is a program that automatically translates another program from some programming language to machine code. The resulting machine instructions are contained in a binary file called an *executable object file* (or simply *executable*). After compiling the source file `hello.c`, the resulting executable `hello` in Figure 1.5 is stored in secondary storage. It is ready to be loaded into the computer's main memory and executed.

## 1.7 Abstractions in Computer Science

*Abstraction* plays a key role in computer science. Computer science is fundamentally a science of abstraction. The concept of abstraction is pervasive in many arts and sciences. Abstraction is the process of considering the external properties of an object independently of its internal details. In other words, by abstracting an object, we are interested in "what the object does" without any interest in "how the object does it." To solve a complex problem, we devise an understandable model for it through abstraction. Then, we explore appropriate methods to solve it using the model. For example, the behavior of electronic circuits used to build computers can be modeled very well by logic gates and Boolean logic. Although this modeling is not exact, the model abstracts away many details, such as the behavior of transistors and electrical signals between

them.

We are able to design, analyze, and manage a complex computer system by applying abstractions. In this case, abstractions are built layer upon layer. As shown in Figure 1.6, the layer of abstraction starts from electronic circuits. Electronic circuits required to build a computer abstract away the requirements of the particular solid-state device technology (e.g., CMOS, NMOS, gallium arsenide, etc.) used to build the circuit. Then, as mentioned before, the behavior of the electronic circuits is modeled with logic gates and Boolean logic. At the next layer, each component of computer hardware (e.g., ALU, CU, and main memory) is implemented with logic gates and Boolean logic. There are various choices of hardware component implementations. They differ in performance, power consumption, and cost. However, at the computer hardware level, we do not worry about such implementation details.

An *instruction set architecture* (ISA) is the complete specification of the interface between the programmer and the underlying computer hardware. The ISA, not the computer hardware, is visible to the programmer when the programmer writes a program. An ISA is implemented by a *microarchitecture* that describes how the ISA should be implemented. Similar to the hardware component implementations with logic gates, microarchitectures have trade-offs between performance, power consumption, and cost. For example, the x86 ISA was originated by Intel and has been implemented by microprocessors from both Intel and AMD with radically different microarchitectures.

On top of the ISA, the assembly language provides one abstraction level. It implements a symbolic representation of machine instructions in the ISA to make the programming easier. An assembler is the interface between the assembly language and the ISA. The assembly language is also abstracted to the C language layer in a similar manner. The C compiler is the interface between the C language and the assembly language. A C program is translated into an assembly language program by the C compiler. High-level languages like C make writing a program simpler and easier than a low-level language, such as machine and assembly languages. Moreover, programs written in high-level languages can be transferred from one computer to another with little modification as long as the target computer has a compiler for the language.

An *algorithm* is a sequence of steps for carrying out a task or solving a problem. Each step is precisely and unambiguously specified and can be carried out by a computer. An algorithm is expressed formally as programs written in programming languages. As shown in Figure 1.6, an algorithm can be implemented with a C program.



## CHAPTER 2

---

# Number Systems

---

Inside computers, information is encoded as patterns of bits because it is easy to construct electronic circuits that exhibit the two alternative states, 0 and 1. The meaning of bits depends on the interpretation. Since the information is processed by the computer essentially in some numerical form, we explore the ways how a computer represent numeric data internally in this chapter.

### 2.1 Positional Number Systems

A *number system* is a system of representing numbers. It is defined by a set of basic symbols called *digits* or *numerals*, and the ways in which the digits can be combined to represent the numbers in the system. A number system can represent integers, fractions, or mixed numbers. A *mixed number* has two parts: an integer part that tells you the whole and a fraction part that is less than one whole. A *radix point* separates the integer part and the fraction part.

Since the decimal number system is the most familiar number system and used in our everyday life, our discussion begins with it. The *decimal number system* is a *positional number system*. In a positional number system, a number is represented by a string of digits. The value of each digit in the string is determined by the position it occupies in the number. That is, each digit position has a weight associated with it. The rightmost position in the number has the lowest weight. The leftmost digit in the number is called the *most significant digit* (MSD) and the rightmost digit is called the *least significant digit* (LSD). Other examples of commonly used positional number systems include binary, octal, and hexadecimal number systems.

In the decimal number system, there are 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The decimal number of the form  $d_{p-1}d_{p-2}\dots d_1d_0.d_{-1}d_{-2}\dots d_{-n}$  has the

Name	Base	Digits
Binary	2	0, 1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Table 2.1: Commonly used number systems with their bases and digit sets.

value,

$$\sum_{i=-n}^{p-1} d_i \cdot 10^i$$

All the weights in the decimal system are powers of the number 10. The integer and fraction parts are separated by the . symbol, the *decimal point*. The fraction part is denoted by a sequence of digits whose weights are negative powers of 10. The decimal number system is also known as the *base-10* number system because the weight of each position in the number is a power of 10. For example, the decimal number 754.82 can be decomposed as follows:

$$754.82 = 7 \times 10^2 + 5 \times 10^1 + 4 \times 10^0 + 8 \times 10^{-1} + 2 \times 10^{-2}$$

When we replace the base 10 with some other whole number  $r$ , called the *base* or *radix*, then we have *base- $r$  number system*. This number system requires  $r$  distinct digits, and the weight in position  $i$  is  $r^i$ . Thus, the base- $r$  number of the form  $x_{p-1}x_{p-2}\dots x_1x_0.x_{-1}x_{-2}\dots x_{-n}$  has the value,

$$\sum_{i=-n}^{p-1} x_i \cdot r^i$$

where each  $x^i$  comes from the set of  $r$  distinct digits. Traditionally, the first  $r$  decimal digits serve as the digits for the base- $r$  systems when  $r \leq 10$ . When  $r > 10$ , the first  $r - 10$  uppercase letters of the alphabet are used to provide additional symbols.

For example, the three-digit number 101 denotes five in the binary number system whereas it denotes one hundred and one in the decimal system. When the base of a base- $r$  number  $N$  is unclear from the context, we append a subscript  $r$  to it. For example,  $101_2$  represent a binary number 101. Table 2.1 shows the bases and digit sets for some commonly used number systems.

## 2.2 Scientific Notation

*Scientific notation* is a scheme of representing decimal numbers that are very small or vary large. In scientific notation, a number is represented in the form:

$$x \times 10^y$$



Where  $x$  is called the *coefficient* (also called the *significand* or *mantissa*) and is any real number, and  $y$  is called the *exponent* and must be an integer. For example,  $-320000000$  can be written as  $-32.0 \times 10^7$  in scientific notation.

Since there are many ways to represent a number in the form of  $x \times 10^y$ , we adopt the convention of making the significand,  $x$ , always in the range:  $1 \leq |x| < 10$ . This notation is often referred to as *normalized scientific notation*. Note that we can not express zero with the normalized scientific notation.

In a positional number system, the *significant digits* of a number are those digits whose removal changes the numerical value associated with the number. A number's *precision* or *accuracy* is the number of significant digits it contains. *Leading zeroes* of a number are any consecutive zeroes that appear in the leftmost positions of the number's representation. On the other hand, *trailing zeroes* are any consecutive zeroes in the number's representation after which no other digits follow. Leading zeroes are insignificant and do not affect the numerical value. Similarly, trailing zeroes that appears to the right of a radix point do not affect the value. They are merely placeholders to indicate the size of the representation. For example, the removal of leading and trailing zeroes in  $0003.1400_{10}$  does not affect the value of the representation.

Base- $r$  numbers expressed with a fixed number of digits often have an *implicit* radix point at some *fixed* position. For example, if the number is an integer, the radix point is immediately to the right of its LSD. If it is a fraction, the radix point is immediately to the left of its MSD. These numbers are referred to as *fixed-point numbers*. In contrast, *floating-point numbers* have a radix point that can be placed anywhere relative to their significant digits: the radix point can *float*. The position of the radix point is indicated separately and encoded in the number's representation. Scientific notation is closely related to the floating-point numbers. We will describe later how the computers express floating-point numbers internally.

## 2.3 Binary Number System

Since it was easier to build electronic circuits that distinguish between two different values than more than two values, computers use *Boolean logic*, which is a two-valued logical system, to abstract away the details of electronic circuits. Consequently, the binary number system are directly related to the Boolean logic used in the computer, and the computer internally represents numeric data in a binary form.

As shown in Table 2.1, the binary number system uses 0 and 1 as its digits. The general form of a binary number is  $b_{p-1}b_{p-2}...b_1b_0.b_{-1}b_{-2}...b_{-n}$ , and it has the following interpretation,

$$\sum_{i=-n}^{p-1} b_i \cdot 2^i$$

For example,

$$10011_2 = 1 \cdot 2^4 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

Binary	Octal	Hexadecimal	Decimal
0000	00	0	0
0001	01	1	1
0010	02	2	2
0011	03	3	3
0100	04	4	4
0101	05	5	5
0110	06	6	6
0111	07	7	7
1000	10	8	8
1001	11	9	9
1010	12	A	10
1011	13	B	11
1100	14	C	12
1101	15	D	13
1110	16	E	14
1111	17	F	15

Table 2.2: The correspondence between 4-bit binary, octal, hexadecimal, and decimal numbers.

$$101.001_2 = 1 \cdot 2^2 + 1 \cdot 2^0 + 1 \cdot 2^{-3} = 5.125$$

Similar to the decimal number system, the leftmost bit  $b_{p-1}$  is called the *most significant bit* (MSB) and the rightmost bit  $b_{-n}$  is called the *least significant bit* (LSB). The radix point in the binary number system is referred to as the *binary point*.

The octal number system is useful for representing multi-bit binary numbers. Since eight is a power of two ( $8 = 2^3$ ), three bits in a binary number can be uniquely represented with a single octal digit. For example,

$$100011001110_2 = 100\ 011\ 001\ 110_2 = 4316_8$$

$$10.1011001011_2 = 010\ .\ 101\ 100\ 101\ 100_2 = 2.5454_8$$

Similarly, the hexadecimal number system is also useful for representing multi-bit binary numbers. Each group of four bits in a binary number can be uniquely represented by a single hexadecimal digit. For example,

$$100011001110_2 = 1000\ 1100\ 1110_2 = 8CE_{16}$$

$$10.1011001011_2 = 0010\ .\ 1011\ 0010\ 1100_2 = 2.B2C_{16}$$

Table 2.2 shows the correspondence between 4-bit binary, octal, hexadecimal, and decimal numbers.

Let  $N$  be the decimal number and  $b_{n-1}b_{n-2} \dots b_0$  be the binary number after the conversion.

1. Set  $i$  to 0.
2. Divide  $N$  by 2 and obtain a quotient and a remainder.
3. Set  $b_i$  to the remainder and  $N$  to the quotient.
4. If  $N$  is not zero, set  $i$  to  $i+1$  and go to step 2.

Figure 2.1: A decimal to binary conversion algorithm for an integer.

$i$	$N$	$b_i$
0	$108/2 = 54$	0 ( <i>LSB</i> )
1	$54/2 = 27$	0
2	$27/2 = 13$	1
3	$13/2 = 6$	1
4	$6/2 = 3$	0
5	$3/2 = 1$	1
6	$1/2 = 0$	1 ( <i>MSB</i> )

Figure 2.2: Converting  $108_{10}$  to binary.

A binary number system using  $n$  bits can represent  $2^n$  different numbers. When such a fixed precision binary number is used to represent only positive values, it is called an *unsigned number*. It is also possible to encode *signed numbers* (positive numbers and negative numbers) in binary. There are several ways to represent the signed numbers in binary. One way of representing a negative number is setting the MSB to 1 and using the remaining bits to represent the value. In this case, the MSB is referred to as the *sign bit*. However, to keep the computer hardware implementation as simple as possible, almost all today's computers internally use *twos complement representation*.

## 2.4 Number System Conversion

Since humans typically use decimal numbers, but computers use binary numbers internally, it is important to know how to convert decimal numbers to binary numbers and *vice versa*. An integer  $N$  can be converted from decimal form to binary form by repeatedly dividing  $N$  by two and using the remainders. Figure 2.1 shows an algorithm for such a conversion. For example, a positive integer  $108_{10}$  is converted to the binary number  $1101100_2$ . The conversion process that follows the algorithm described in Figure 2.1 is illustrated in Figure 2.2.

Let  $N$  be the decimal fraction and  $b_{-1}b_{-2} \dots b_{-n}$  be the binary number after the conversion.

1. Set  $i$  to 1.
2. Multiply  $N$  by 2 and set  $N$  to the result.
3. If  $N$  is less than 1, set  $b_{-i}$  to 0.
4. Otherwise, set  $b_{-i}$  to 1 and  $N$  to  $N-1$ .
5. If  $i$  is less than  $n$ , set  $i$  to  $i+1$  and go to step 2.

Figure 2.3: A decimal to binary conversion algorithm for fractions.

$i$	$N$	$b_{-i}$
1	$0.735 \times 2 = 1.47$	1
2	$0.47 \times 2 = 0.94$	0
3	$0.94 \times 2 = 1.88$	1
4	$0.88 \times 2 = 1.76$	1

Figure 2.4: Converting  $0.731_{10}$  to a 4-bit binary fraction.

For a mixed number, we convert its integer part and fraction part to binary form separately. Then, we combine the results. An algorithm for converting a decimal fraction to binary form is shown in Figure 2.3. To obtain the first  $n$  bits of a binary fraction using the algorithm, it is required to perform  $n$  multiplications by two in general. For example, the process of converting a decimal fraction  $0.731_{10}$  to a 4-bit binary fraction  $0.1011_2$  is illustrated in Figure 2.4. After combining the results from the integer conversion and the fraction conversion, a mixed number  $108.731_{10}$  in the above examples is converted to a binary number  $1101100.1011_2$ .

However, the 4-bit binary fraction obtained in the process of Figure 2.4 is inexact. So far, we assumed that there are as many bits available as required to represent the number, and we did not consider the size of the representation. When the number of bits used is specified to represent a number, it determines the range of possible numbers that can be represented. For example, 8 bits can represent  $256 = 2^8$  distinct numeric values. *Word* is a group of bits that are handled as a unit by the computer. The number of bits in a word is an important characteristic of a computer architecture. The precision of expressing a numerical quantity strongly depends on the word size (i.e., the number of bits in a word). If the word size is only four bits, then we can not express the result of the above example more precisely.

A *dyadic fraction* is a rational number whose denominator is a power of two;

$i$	$N$	$b_i$
0	$108/8 = 13$	4 ( <i>LSB</i> )
1	$13/8 = 1$	5
2	$1/8 = 0$	1 ( <i>MSB</i> )

$i$	$N$	$b_{-i}$
1	$0.735 \times 8 = 5.88$	5
2	$0.88 \times 8 = 7.04$	7
3	$0.04 \times 8 = 0.32$	0
4	$0.32 \times 8 = 2.56$	2

Figure 2.5: Converting  $108.731_{10}$  to an octal number.

$i$	$N$	$b_i$
0	$108/16 = 6$	12( <i>C</i> ) ( <i>LSB</i> )
1	$6/16 = 0$	6 ( <i>MSB</i> )

$i$	$N$	$b_{-i}$
1	$0.735 \times 16 = 11.76$	11( <i>B</i> )
2	$0.76 \times 16 = 12.16$	12( <i>C</i> )
3	$0.16 \times 16 = 2.56$	2
4	$0.56 \times 16 = 8.96$	8

Figure 2.6: Converting  $108.731_{10}$  to a hexadecimal number.

for example,  $1/2$  or  $3/16$ , but not  $1/3$ . Dyadic decimal fractions convert to finite binary fractions and are represented in full precision if the word size is greater than or equal to the length of the binary representation. Non-dyadic decimal fractions convert to infinite binary fractions (a repeating sequence of the same bit pattern). *Truncation* (also known as *chopping*) is the process of discarding any unwanted digits of a number. *Rounding* a number replaces the number with another number that is as close to the original number as possible.

Decimal to octal and decimal to hexadecimal conversions are similar to the decimal to binary conversion. An integer can be converted from decimal form to octal (or hexadecimal) form by repeatedly dividing it by eight (or sixteen) and using the remainders. A decimal fraction converts an octal (or hexadecimal) fraction with  $n$  digits after performing  $n$  successive multiplications by eight (or sixteen). For example, the conversion process of a mixed number,  $108.731_{10}$ , to octal ( $154.5702_8$ ) and hexadecimal ( $6C.BC28_{16}$ ) are illustrated in Figure 2.5 and Figure 2.6, respectively.

Another way to perform decimal to octal or decimal to hexadecimal conversion for a decimal number is that converting it to a binary number. Then, the binary number can be converted to an equivalent octal or hexadecimal number. For example,  $108.731_{10}$  is converted to octal ( $154.5702_8$ ) and hexadecimal ( $6C.BC28_{16}$ ) in the following way:

$$\begin{aligned}
108.731_{10} &= 001\ 101\ 100 . 101\ 111\ 000\ 010_2 \\
&= 154.5702_8 \\
108.731_{10} &= 0110\ 1100 . 1011\ 1100\ 0010\ 1000_2 \\
&= 6C.BC28_{16}
\end{aligned}$$

## 2.5 Unsigned Binary Arithmetic

Arithmetic operations performed on unsigned binary numbers are much like arithmetic in the decimal number system that we know very well. Addition, subtraction, multiplication, and division can be performed on binary numbers. A pair of unsigned binary numbers can be added bit by bit according to the same method used in decimal addition. Since computers use different standard to encode a mixed number (typically using the IEEE 754 standard), we will restrict our discussion to unsigned whole numbers.

For example, a pair of unsigned 4-bit binary numbers 1001 and 0101 can be added as follows:

$$\begin{array}{r}
 \text{carry} \quad 0 \quad 0 \quad 0 \quad 1 \\
 \phantom{\text{carry}} \quad 1 \quad 0 \quad 0 \quad 1 \\
 + \phantom{\text{carry}} \quad 0 \quad 1 \quad 0 \quad 1 \\
 \hline
 \phantom{\text{carry}} \quad 0 \quad 1 \quad 1 \quad 1 \quad 0
 \end{array}$$

When adding two 1s in the same column produces  $10_2$ , 1 (called *carry*) will have to be added to the left column of more significant bits. When the result of addition exceeds one, we carry the excess amount to the next position using a carry. The position has a weight that is higher by a factor equal to two.

An *overflow* occurs when a computation produces a value that falls outside the range of values that can be represented. For example, adding two unsigned 4-bit binary numbers 1011 and 0111 produces a 5-bit value 10010, and it is an overflow:

$$\begin{array}{r}
 \text{carry} \quad 1 \quad 1 \quad 1 \quad 1 \\
 \phantom{\text{carry}} \quad 1 \quad 0 \quad 1 \quad 1 \\
 + \phantom{\text{carry}} \quad 0 \quad 1 \quad 1 \quad 1 \\
 \hline
 \phantom{\text{carry}} \quad 1 \quad 0 \quad 0 \quad 1 \quad 0
 \end{array}$$

The carry bit from the MSB tell us that an overflow occurred in the addition.

Unsigned binary subtraction works in much the same way as decimal subtraction. For example, 0101 can be subtracted from 1001 produces 0100 as follows:

$$\begin{array}{r}
 \text{borrow} \quad 0 \quad 1 \quad 0 \quad 0 \\
 \phantom{\text{borrow}} \quad 1 \quad 0 \quad 0 \quad 1 \\
 - \phantom{\text{borrow}} \quad 0 \quad 1 \quad 0 \quad 1 \\
 \hline
 \phantom{\text{borrow}} \quad 0 \quad 0 \quad 1 \quad 0 \quad 0
 \end{array}$$

Subtracting a 1 from a 0 in a column produces 1 by borrowing 1 (called *borrow*) from the left column of more significant bits. When the result of a bit by bit subtraction in a column is less than 0, we borrow 1 from the column on the left subtracting it from the next higher positional value.

Unsigned binary multiplication is also similar to the decimal multiplication that adds up partial products to produce the final result. For example, two

Positive		Negative	
Ones' complement	Decimal	Ones' complement	Decimal
0000	0	1111	-0
0001	1	1110	-1
0010	2	1101	-2
0011	3	1100	-3
0100	4	1011	-4
0101	5	1010	-5
0110	6	1001	-6
0111	7	1000	-7

4-bit unsigned binary numbers 1001 and 0101 are multiplied as follows:

$$\begin{array}{cccccccc} & & & & 1 & 0 & 0 & 1 \\ \times & & & & 0 & 1 & 0 & 1 \\ \hline & & & & 1 & 0 & 0 & 1 \\ & & & 0 & 0 & 0 & 0 & \\ & & 1 & 0 & 0 & 1 & & \\ & 0 & 0 & 0 & 0 & & & \\ \hline & 0 & 1 & 0 & 1 & 1 & 0 & 1 \end{array}$$

Unsigned binary division is again similar to the decimal division. For example, the process of dividing 11101 by 0100 produces a quotient 111 and a remainder 1.

[illegible]

*Ones' complement representation* is one of the methods to represent a negative number. The ones' complement of a negative binary number is the bitwise inversion (flipping 0's for 1's and *vice-versa*) of its positive counterpart. For example, the ones complement representation of  $-6$  ( $-0110_2$ ) is  $1001_2$ .

Copyright ©2013 by Jaejin Lee

the numbers that can be represented in the ones' complement representation with 4 bits. The MSB is the sign bit. Unlike two's complement representation (explained later), it has two zeroes. For example,  $0000_2$  and  $1111_2$  are the two 4-bit ones' complement representations of zero.

Adding two binary numbers in the ones' complement representation is similar to the unsigned binary addition, but there is a condition called an *end-around carry* that does not occur in unsigned binary addition. Consider adding two binary numbers 0100 and 1110 in ones' complement representation:

$$\begin{array}{r}
 \text{carry } 1 \quad 1 \quad 0 \quad 0 \\
 \quad \quad 0 \quad 1 \quad 0 \quad 0 \quad (4) \\
 + \quad \quad 1 \quad 1 \quad 1 \quad 0 \quad (-1) \\
 \hline
 \quad \quad 0 \quad 0 \quad 1 \quad 0 \\
 \quad \quad \quad \quad 1 \quad (\text{add the end-around carry}) \\
 \hline
 \quad \quad 0 \quad 0 \quad 1 \quad 1 \quad (3)
 \end{array}$$

For the addition in the ones' complement representation, we perform a unsigned binary addition. If there is a carry from the MSB (this is called an *end-around carry*), then add the carry back into the resulting sum to produce the final result.

Unlike the two's complement representation, the ones' complement representation is not popular in these days because of several issues like negative zero, end-around carry, etc.

## 2.7 Two's Complement Representation

The two's complement representation is the most common method of representing signed integers internally in computers. It simplifies the complexity of the ALU by allowing using the same circuit that is used to implement arithmetic operations for unsigned numbers. The major difference is the interpretation of the result produced by the arithmetic operation.

To convert a number to two's complement representation, the number is converted to its ones' complement first. Then one is added to the result to produce its two's complement. Another way of encoding a negative number to its two's complement is flipping all bits but the first least significant 1 and all the trailing 0s. For example, the two's complement representation of 6 ( $0110_2$ ) is  $1010_2$ .

The maximum value that can be represented by  $n$ -bit two's complement representation is  $2^{n-1} - 1$  and the minimum is  $-2^{n-1}$ . Table 2.4 shows the numbers that can be represented in the two's complement representation with 4 bits. The MSB is the sign bit.

Two's complement addition is exactly the same as that of unsigned binary numbers. For example, adding 4 ( $0100_2$ ) and 5 ( $1011_2$ ) in 4-bit two's complement representation gives 1 ( $1111_2$ ). Subtraction can be handled by converting it to addition:

$$x - y = x + (-y)$$



Positive			
Two's complement	Decimal	Negative	
		Two's complement	Decimal
0000	0		
0001	1	1111	-1
0010	2	1110	-2
0011	3	1101	-3
0100	4	1100	-4
0101	5	1011	-5
0110	6	1010	-6
0111	7	1001	-7
		1000	-8

Table 2.4: 4-bit two's complement representation.

1. Add the two operands including the sign bits.
2. If the carry into the MSB is not equal to the carry out of the MSB, an overflow has occurred.

Figure 2.7: Overflow detection in two's complement addition.

For example, an overflow occurs in the following 4-bit two's complement addition:

$$\begin{array}{rcccccl} \text{carry} & 0 & 1 & 1 & 1 & \\ & & 0 & 1 & 1 & 1 \quad (7) \\ + & & 0 & 1 & 1 & 1 \quad (7) \\ \hline & 0 & 1 & 1 & 1 & 0 \end{array}$$

In this case, the carry into the MSB (1) is different to the carry (0) out of the MSB.

## 2.8 Shift Operations and Sign Extension

A shift operation shifts all of the bits in its operand. Every bit in its operand is moved a given number of positions to a specified direction. There two different types of *shift* operations: *logical shift* and *arithmetic shift*.

A logical shift operation moves bits to the left or right. The bits that fall off at the end of the word are discarded. The vacant positions in the opposite

	$x \gg 3$	$x \ll 3$
Logical shift	00010011	11101000
Arithmetic shift	11110011	11101000

Figure 2.8: The difference between 8-bit logical shift and arithmetic shift for  $x = 10011101$ .

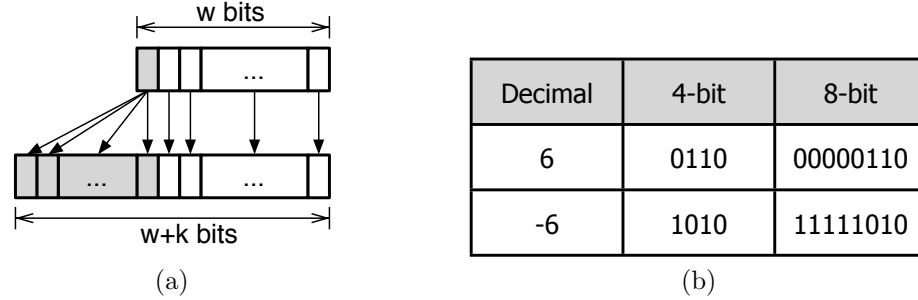


Figure 2.9: (a) The sign extension process. (b) The sign-bit repetition for -6.

end are filled with 0s. The arithmetic left shift is the same as the logical left shift. In the arithmetic right shift, the leftmost bits are filled with the sign bit of the original number. We denote a *left shift* operation with  $\ll$  and *right shift* operation with  $\gg$ . Figure 2.8 shows an example of the difference between logical shift and arithmetic shift for  $x = 10011101$ .

Using shift operations, we can efficiently perform unsigned multiplication or division by powers of two. An  $n$ -bit logical left shift operation on unsigned integers is equivalent to multiplication by  $2^n$ . For example,

$$00001011_2 (13) \ll 2 = 13 \times 2^2 = 00101100_2 (52)$$

An  $n$ -bit logical right shift is equivalent to division by  $2^n$ , and we obtain the quotient of the division. For example,

$$00101101_2 (53) \gg 2 = 53 \div 2^2 = 00001011_2 (13)$$

Using arithmetic shifts, we can efficiently perform multiplication or division of signed integers by powers of two. In two's complement representation, arithmetic shift operations extend the notion of the *floor* operation. The floor of an integer  $x$  (denoted by  $\lfloor x \rfloor$ ) is defined by the greatest integer less than or equal to  $x$ . For example,  $\lfloor 11/2 \rfloor = 5$  and  $\lfloor -3/2 \rfloor = -2$ . An arithmetic shift operation on an integer takes the floor of the result of multiplication or division. For example,

$$\begin{aligned} 1101_2 \quad (-4) \ll 1 &= 1010_2 \quad (-6) \\ 1101_2 \quad (-4) \gg 1 &= 1110_2 \quad (-2) \\ 1101_2 \quad (-4) \gg 2 &= 1111_2 \quad (-1) \\ 1100_2 \quad (-4) \ll 2 &= 0000_2 \quad (0) \quad \text{Overflow} \\ 0100_2 \quad (4) \ll 1 &= 1000_2 \quad (-8) \quad \text{Overflow} \end{aligned}$$

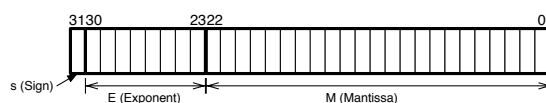


Figure 2.10: The single-precision (32-bit) IEEE floating-point format.

However, in two's complement representation, an arithmetic right shift is not equivalent to division by a power of two. For example, when you perform an arithmetic right shift on a 4-bit  $-1 = 1111_2$ , you still get  $-1 = 1111_2$ . The ANSI C99 standard does not specify the definition of the C language's right shift operator for negative values. The behavior of the right shift operator depends on the C compiler.

When converting an integer with  $w$  bits into the one with  $w + k$  bits and the same value, we need to perform *sign extension* in two's complement representation. The MSB (the sign bit) must be repeated in all the  $k$  extra bits. Figure 2.9(a) shows this process and Figure 2.9(b).

## 2.9 Floating-Point Representation

IEEE 754 Standard[?, ?, ?] is established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE) as a uniform standard for floating point arithmetic. It has been widely adopted by almost all major CPU vendors. Before IEEE 754 Standard, there have been many incompatible floating-point representations introduced by different computer manufacturers. The IEEE 754 standard published in 1985 was superseded in 2008 by the IEEE 754-2008 standard.

The IEEE floating-point representation of a number has the following form:

$$N = (-1)^s \times M \times 2^E$$

The sign bit  $s$  determines if the number is negative ( $s = 1$ ) or positive ( $s = 0$ ). The significand (or mantissa)  $M$  is a fractional value and  $1.0 \leq M < 2.0$ . Exponent  $E$  weights the value by power of two. The bit representation of the single-precision (32-bit) version of the IEEE format is shown in Figure 2.10.

Encoding MSB is sign bit exp field encodes E frac field encodes M Single precision (32 bits): 8 exp bits, 23 frac bits Double precision (64 bits): 11 exp bits, 52 frac bits Representing normalized values, denormalized values, +, -, NaN Normalized Numeric Values (optional) When exp 0000 and exp 1111 Exponent coded as biased value  $E = \text{Exp} - \text{Bias}$  Exp : unsigned value denoted by exp Bias : Bias value Single precision: 127 (Exp: 1254, E: -126127) Double precision: 1023 (Exp: 12046, E: -10221023) In general:  $\text{Bias} = 2^{e-1} - 1$ , where  $e$  is the number of exponent bits Significand coded with implied leading 1 Extra leading bit for free  $M = 1.\text{xxxx}2^{-\text{xxxx}}$  xxxx: bits of frac Minimum when 0000 ( $M = 1.0$ ) Maximum when 1111 ( $M = 2.0$ ) For example, Value 14324.5 = 11 0111 1111 0100.12 = 1.1011 1111 1010 012 X 213 Significand  $M = 1.1011 1111$

1010 012 frac = 1011 1111 1010 0100 0000 0002 Exponent E = 13 Bias = 127 exp = 140 = 1000 11002 IEEE 754 floating point representation 0100 0110 0101 1111 1101 0010 0000 00002

Denormalized Values (optional) Condition exp = 0000 Value Exponent value E = Bias + 1 Significand value M = 0.xxxx2 xxxx: bits of frac When exp = 0000, frac = 0000 Represents value 0 +0 and 0 When exp = 0000, frac 0000 Numbers very close to 0.0 Lose precision as get smaller Gradual underflow Special Values (optional) Condition exp = 1111 When exp = 1111, frac = 0000 Represent value (infinity) Operation that overflows Both positive and negative E.g.,  $1.0/0.0 = 1.0/0.0 = +$ ,  $1.0/0.0 = -$  When exp = 1111, frac 0000 Not-a-Number (NaN) Represents case when no numeric value can be determined E.g.,  $\text{sqrt}(1)$ , Summary of Floating Point Encodings (optional)

## CHAPTER 3

---

# Logic Circuits

---

The elementary building blocks of a CPU are *logic gates*. The input and output of logic gates assume only two values, 0 and 1. Their behavior is well modeled by laws of *Boolean algebra*. We start out this chapter with Boolean algebra, and learn how an arithmetic logic unit and main memory are built from logic gates.

### 3.1 Boolean Algebra

*Boolean algebra* was developed by George Boole in 1854. It is an algebraic representation of logic and the algebra of truth values: true and false. Edward V. Huntington gave the formal definition of Boolean algebra in 1904. It is referred to as *Huntington's postulates*. In 1937, Claude Shannon applied Boolean algebra to electrical switching circuits to reason about the behavior of networks of relay switches in 1937. In the rest of this book, the *two-element Boolean algebra* (also known as *switching algebra*) defined by Shannon is used and referred to as Boolean algebra.

The two-element Boolean algebra is a set of two elements,  $B = \{0, 1\}$ , together with three operations  $\vee$  (*OR*),  $\wedge$  (*AND*), and  $'$  (*NOT*). It satisfies the following axioms:

**Axiom 1** (Closure). *For all  $x$  and  $y$  in  $B$  both  $x \vee y$  and  $x \wedge y$  are in  $B$ .*

**Axiom 2** (Identity element). *There exist distinct elements 0 and 1 in  $B$  such that for all  $x$  in  $B$ ,  $x \vee 0 = x$  and  $x \wedge 1 = x$ .*

**Axiom 3** (Commutativity). *For all  $x$  and  $y$  in  $B$ ,  $x \vee y = y \vee x$  and  $x \wedge y = y \wedge x$ .*

**Axiom 4** (Distributivity). *For all  $x$ ,  $y$ , and  $z$  in  $B$ ,  $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$  and  $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ .*

OR			AND			NOT	
x	y	$x \vee y$	x	y	$x \wedge y$	x	$x'$
0	0	0	0	0	0	0	1
0	1	1	0	1	0	1	0
1	0	1	1	0	0		
1	1	1	1	1	1		

Figure 3.1: The three operations in Boolean algebra: OR, AND, and NOT.

x	y	z	$f(x,y,z)$ $= ((x \vee y) \wedge z')$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Figure 3.2: The truth table of a Boolean function  $f(x, y, z) = ((x \vee y) \wedge z')$ .

**Axiom 5** (Complement). *For each  $x$  in  $B$ , there exists an element in  $B$ , denoted  $x'$  and called the complement or negation of  $x$ , such that  $x \vee x' = 1$  and  $x \wedge x' = 0$ .*

**Axiom 6** (Cardinality). *There are at least two distinct elements in  $B$ .*

The three operations, OR ( $\vee$ ), AND ( $\wedge$ ), and NOT ( $'$ ) are defined as follows:

$$\begin{aligned}
 x \vee y &= \begin{cases} 1 & \text{when either } x \text{ or } y \text{ is 1, or both are 1} \\ 0 & \text{otherwise} \end{cases} \\
 x \wedge y &= \begin{cases} 1 & \text{when both } x \text{ and } y \text{ are 1} \\ 0 & \text{otherwise} \end{cases} \\
 x' &= \begin{cases} 1 & \text{when } x = 0 \\ 0 & \text{otherwise} \end{cases}
 \end{aligned}$$

Figure 3.1 defines these operations with truth tables. The *truth table* of a Boolean operation is a list of all combinations of 1s and 0s that can be inputs to the operation and a list that shows the value of the operation.

There are some other properties (i.e., theorems) of Boolean algebra that can be derived from the axioms:

f(x, y)	xy				Comment
	00	01	10	11	
0	0	0	0	0	Constant 0
$x \wedge y$	0	0	0	1	x and y (AND)
$x \wedge y'$	0	0	1	0	x but not y
x	0	0	1	1	x
$x' \wedge y$	0	1	0	0	y but not x
y	0	1	0	1	y
$(x \wedge y') \vee (x' \wedge y)$	0	1	1	0	x or y but not both (XOR)
$x \vee y$	0	1	1	1	x or y (OR)
$(x \vee y)'$	1	0	0	0	NOR
$(x \wedge y) \vee (x' \wedge y')$	1	0	0	1	x equals y (XNOR)
$y'$	1	0	1	0	NOT y
$x \vee y'$	1	0	1	1	If y then x
$x'$	1	1	0	0	NOT x
$x' \vee y$	1	1	0	1	If x then y
$(x \wedge y)'$	1	1	1	0	NAND
1	1	1	1	1	Constant 1

Figure 3.3: The 16 Boolean functions of two input variables,  $x$  and  $y$ 

**Property 1** (Uniqueness of 0 and 1). *0 and 1 are unique.*

**Property 2** (Idempotency). *For all  $x$  in  $B$ ,  $x \vee x = x$  and  $x \wedge x = x$ .*

**Property 3.** *For all  $x$  in  $B$ ,  $x \vee 1 = 1$  and  $x \wedge 0 = 0$ .*

**Property 4** (Absorption). *For all  $x$  and  $y$  in  $B$ ,  $(x \vee y) \wedge x = x$  and  $(x \wedge y) \vee x = x$ .*

**Property 5** (Associativity). *For all  $x$ ,  $y$ , and  $z$  in  $B$ ,  $(x \vee y) \vee z = x \vee (y \vee z)$  and  $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ .*

**Property 6** (The uniqueness of complement). *For all  $x$  in  $B$ ,  $x'$  is unique.*

**Property 7** (Involution). *For all  $x$  in  $B$ ,  $(x')' = x$ .*

**Property 8** (De Morgan's law). *For all  $x$  and  $y$  in  $B$ ,  $(x \vee y)' = x' \wedge y'$  and  $(x \wedge y)' = x' \vee y'$ .*

A *Boolean expression* is a string of symbols involving constants 0 and 1, some variables, and Boolean operations  $\vee$ ,  $\wedge$ , and  $'$ . A Boolean expression in  $n$  variables  $x_1, x_2, \dots$ , and  $x_n$  is defined inductively as follows:

- Each of the symbols 0, 1,  $x_1, x_2, \dots$ , and  $x_n$  is a Boolean expression.
- If  $e_1$  and  $e_2$  are Boolean expressions, so are  $e_1'$ ,  $(e_1 \vee e_2)$  and  $(e_1 \wedge e_2)$ .

XOR			XNOR			NOR			NAND		
x	y	$x \oplus y$	x	y	$x \odot y$	x	y	$(x \vee y)'$	x	y	$(x \wedge y)'$
0	0	0	0	0	1	0	0	1	0	0	1
0	1	1	0	1	0	0	1	0	0	1	1
1	0	1	1	0	0	1	0	0	1	0	1
1	1	0	1	1	1	1	1	0	1	1	0

Figure 3.4: The truth tables of XOR, XNOR, NOR, and NAND.

The following is some examples of Boolean expressions:

$$((x \vee y)' \vee (x' \wedge y))$$

$$(((x \wedge y) \vee (x \wedge z')) \vee (y' \vee y)')$$

For Boolean expressions  $e_1$  and  $e_2$ ,  $e_1'$  is often called as the *negation* of  $e_1$ , and  $(e_1 \vee e_2)$  and  $(e_1 \wedge e_2)$  are called as the *disjunction* and *conjunction* of  $e_1$  and  $e_2$ , respectively. If  $e$  is a Boolean expression in  $n$  variables  $x_1, x_2, \dots$ , and  $x_n$ , then  $e$  defines a *Boolean function* mapping  $B^n$  into  $B$  whose value at  $n$ -tuple  $(a_1, a_2, \dots, a_n)$  is the element of  $B = \{0, 1\}$  obtained by replacing  $x_1$  by  $a_1$ ,  $x_2$  by  $a_2$ ,  $\dots$ , and  $x_n$  by  $a_n$  in  $e$ . A truth table is the simplest way to specify a Boolean function. For example,  $f(x, y, z) = ((x \vee y) \wedge z')$  is a Boolean function defined by Boolean expression  $(x \vee y) \wedge z'$ . This function is also defined by the truth table in Figure 3.2.

The number of different Boolean functions that can be defined over  $n$  binary variables is  $2^{2^n}$ . Table 3.3 shows all 16 functions of two input variables,  $x$  and  $y$ . A set of Boolean operations is *functionally complete* if its members can construct all other Boolean functions over any given set of input variables. We assume that these operations can be applied as many times as needed. A well known complete set of Boolean operations is {AND, OR, NOT}. Some new Boolean operations shown in Figure 3.3 can be composed from these three basic operations.

For two variables  $x$  and  $y$ , the Boolean operations *XOR* ( $\oplus$ ) and *XNOR* ( $\odot$ ) are defined by

$$\begin{aligned} x \oplus y &= (x \wedge y') \vee (x' \wedge y) = \begin{cases} 1 & \text{when } x \neq y \\ 0 & \text{otherwise} \end{cases} \\ x \odot y &= (x \wedge y) \vee (x' \wedge y') = \begin{cases} 1 & \text{when } x = y \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

XOR and XNOR are shorthands for *exclusive or* and *exclusive NOT-OR*, respectively. NOR is defined by  $(x \vee y)'$  and a shorthand for *NOT-OR*. Similarly, NAND is defined by  $(x \wedge y)'$  and a shorthand for *NOT-AND*. {NOR} and {NAND} are also functionally complete. The truth tables of XOR, XNOR, NOR, and NAND are shown in Figure 3.4.






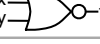



Name	Symbol	Function
NOT		$f = x'$
OR		$f = x \vee y$
AND		$f = x \wedge y$
NOR		$f = (x \vee y)'$
NAND		$f = (x \wedge y)'$
XOR		$f = x \oplus y$
XNOR		$f = x \odot y$

Figure 3.5: The graphic symbols of some elementary logic gates.

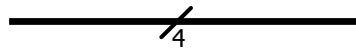
## 3.2 Logic Gates

A *logic gate* is a conceptual or physical device that performs one or more Boolean operations. A Boolean function can be implemented with a logic gate. Logic gates are typically made from *transistors*. However, we focus on the abstract notion of logic gates; we do not worry about their implementation details or how electronic circuits implement them. A logic gate can be viewed as a black box device. If a Boolean function  $f$  maps  $B^n$  into  $B^m$  (i.e., it has  $n$  input variables and  $m$  outputs), the logic gate that implements  $f$  has  $n$  input pins and  $m$  output pins. The graphic symbols of some elementary logic gates are shown in Figure 3.5. A *logic diagram* is a graphical representation of a logic circuit that shows connections between logic gates.

A logic gate with more complicated functionality can be implemented by combining and interconnecting some elementary logic gates. For example, Figure 3.6 shows the implementations of XOR, XNOR, NOR, and NAND gates using AND, OR, and NOT gates. An XOR gate can be implemented with an OR gate, two AND gates, and two NOT gates.

Multi-input gates can also be made by combining gates of the same type with less inputs. Figure 3.7 shows how a 3-input AND gate can be made out of 2-input AND gates.

As mentioned before, a word is a group of bits that are handled as a unit by the computer. Thus, computer hardware is typically designed to handle multiple bits simultaneously. A *bus* is a collection of two or more related signal lines each of which represents a bit. To avoid drawing multiple wires for a bus in logic circuit diagrams, we use the *bus notation*. For example, the following diagram



shows a 4-bit bus:

To refer to individual bits in an  $n$ -bit bus named *data*, we use the notation

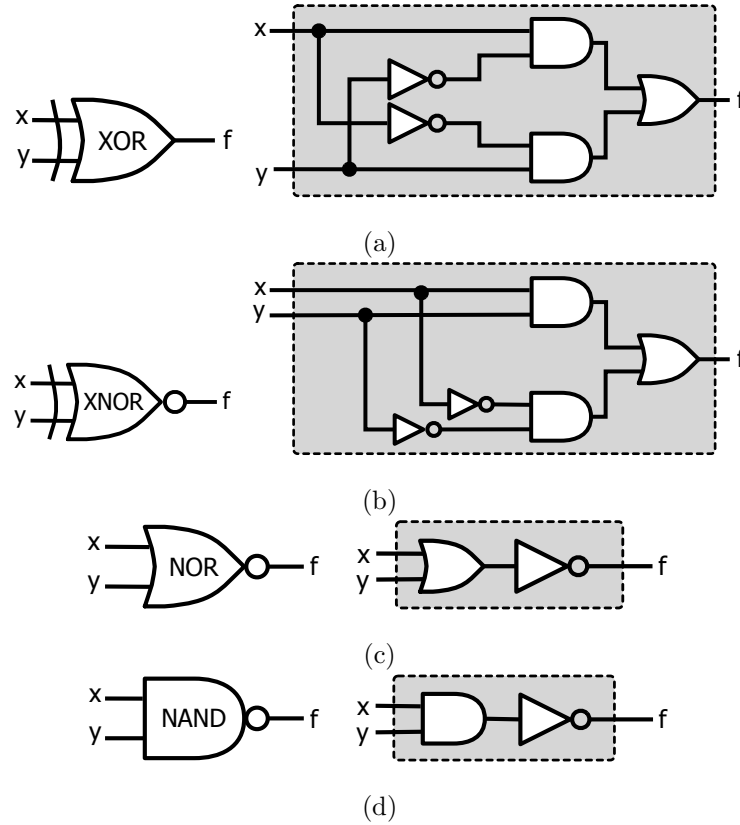
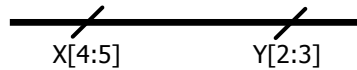


Figure 3.6: Implementations of XOR, XNOR, NOR, and NAND gates with AND, OR, and NOT gates: (a) the logic diagram of XOR; (b) the logic diagram of XNOR; (c) the logic diagram of NOR; (d) the logic diagram of NAND.

$data[i]$  to refer to the bit located at position  $i$  (the LSB is  $data[0]$ ). In addition, to indicate a set of consecutive bits in  $data$ , we use the notation  $data[i : j]$  for  $j - i + 1$  consecutive bits from position  $i$  to  $j$  inclusively. For example, the following diagram shows that  $X[4]$  connects to  $Y[2]$  and  $X[5]$  connects to  $Y[3]$ :



Like Boolean operations, a set of logic gates that can implement any Boolean function is called a *complete* set of logic gates. Each of  $\{\text{AND}, \text{OR}, \text{NOT}\}$ ,  $\{\text{NAND}\}$ , and  $\{\text{NOR}\}$  is a complete set of logic gates. A *universal gate* is a gate that can implement any Boolean function without need to use any other gate type. Thus, NAND or NOR is a universal gate. Since the implementation of NAND or NOR gate requires fewer transistors and is faster than that of AND or OR gates, logic designers prefer to use NAND or NOR gate to implement their logic circuits.

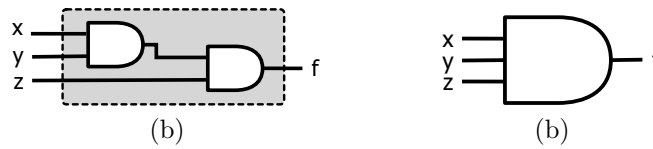


Figure 3.7: An implementation of a 3-input AND gate with two 2-input AND gates: (a) its Logic diagram; (b) its graphic symbol.

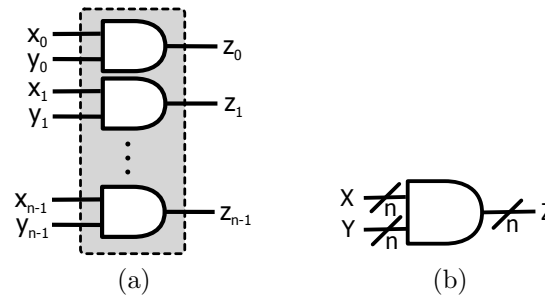
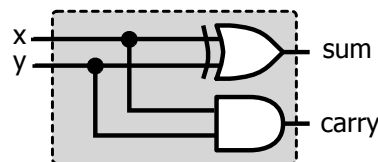


Figure 3.8: An implementation of a  $n$ -bit AND gate with  $n$  2-input AND gates: (a) its logic diagram; (b) its graphic symbol.

A multi-bit ( $n$ -bit) logic gate that performs a bit-wise Boolean operation is implemented by an array of  $n$  gates each operating separately on each bit position of the operands. For example, as shown in Figure 3.8, to implement an  $n$ -bit AND gate that performs a bit-wise AND operation on two  $n$ -bit binary values, we can build an array of  $n$  two-input AND gates each operating separately on a pair of bits from the two operands.

x	y	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a)



(b)

Figure 3.9: A half adder: (a) its truth table; (b) its logic diagram.

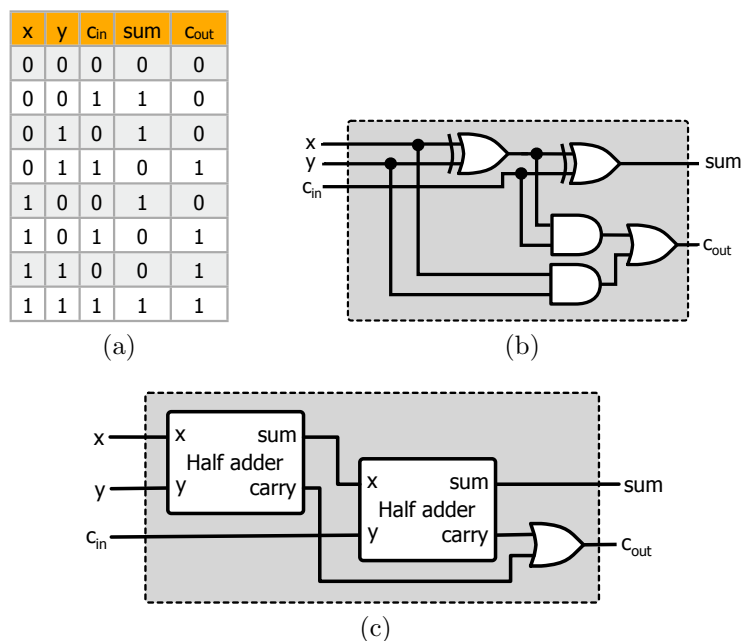


Figure 3.10: A full adder: (a) its truth table; (b) its logic diagram; (c) an implementation with two half adders.

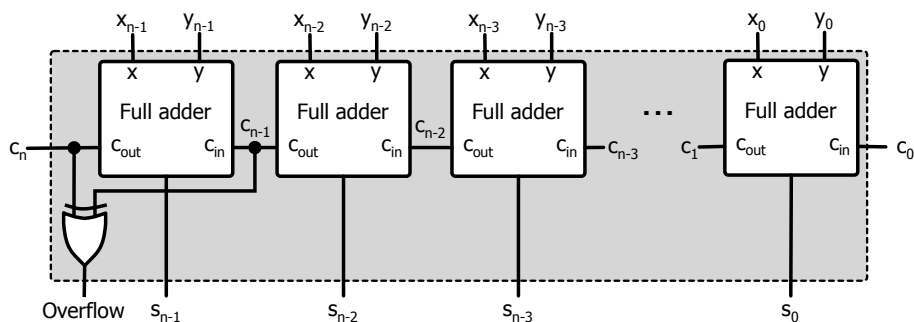


Figure 3.11: An  $n$ -bit ripple carry adder.

### 3.3 Combinational Logic Circuits

A *combinational logic circuit* consists of logic gates and performs an operation that can be specified by a set of Boolean functions. Its outputs are totally dependent on the current input values and determined by combining the input values using Boolean operations. In contrast, the outputs of a *sequential logic*

*circuit* depend not only on the current input values but also on the past inputs. In addition to using logic gates, it employs *memory* and its outputs are a function of the current input values and the data stored in memory. Consequently, a sequential logic circuit has *states*.

### The Half Adder

A *half adder* adds two one-bit binary numbers  $x$  and  $y$ . It has two outputs: sum and carry. The Boolean functions for the sum and carry are given as follows:

$$\begin{aligned}\text{sum} &= x \oplus y \\ \text{carry} &= x \wedge y\end{aligned}$$

Figure 3.9 shows the truth table and logic circuit diagram of a half adder. When we add two multi-bit numbers, the half adder has little practical value. All but LSB columns need to consider the carry from the next least significant position in addition to the two bits from the operands. Only the LSB can be added with a half adder.

### The Full Adder

A *full adder* adds three one-bit binary numbers  $x$ ,  $y$ , and a carry ( $c_{in}$ ). It has two outputs: sum and carry ( $c_{out}$ ). The Boolean functions for the sum and carry are given as follows:

$$\begin{aligned}\text{sum} &= (x \oplus y) \oplus c_{in} \\ \text{carry} &= (x \wedge y) \vee (c_{in} \wedge (x \oplus y))\end{aligned}$$

Two half adders can be combined with an OR gate to make a full adder. The one-bit full adder's truth table and logic diagram are shown in Figure 3.10.

We can implement an  $n$ -bit binary adder by cascading  $n$  full adders. The carry output  $c_{out}$  of the full adder at the next least significant position is connected to the carry input  $c_{in}$  of the current full adder. We call this type of  $n$ -bit adder as a *ripple carry adder* because each carry ripples to the next full adder. The  $n$ -bit ripple carry adder adds two  $n$ -bit binary numbers  $X$  and  $Y$ . The outputs are sum and carry ( $c_n$ ) from the MSB. Figure 3.11 shows the logic diagram of an  $n$ -bit ripple-carry-adder. In the worst case, the carry propagates from the LSB through the MSB. *Gate delay* (also known as *propagation delay*) is the time delay between the changes when an input change causes an output change. The ripple carry adder requires one propagation delay time for each bit position of the operands. Thus, it is not appropriate for high speed computing systems.

The overflow detection logic for two's complement addition is also shown in Figure 3.11. When adding two two's complement binary numbers, as mentioned before in Chapter 2, detecting overflow is done by comparing the carry out of the MSB with the carry into the MSB:

$$\text{Overflow} = c_n \oplus c_{n-1}$$

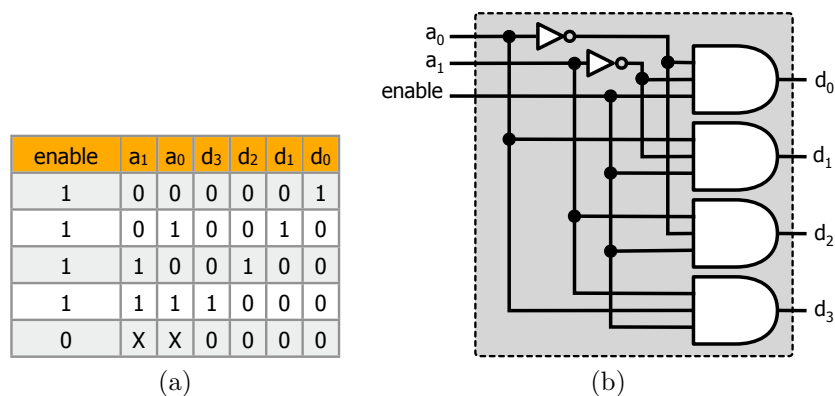


Figure 3.12: A 2-to-4 decoder: (a) its truth table; (b) its logic diagram.

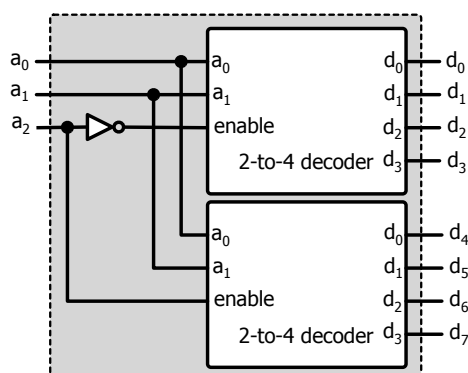


Figure 3.13: Building a 3-to-8 decoder using two 2-to-4 decoders each with an enable input.

## Decoders

A *decoder* (also called *demultiplexer*) converts binary information from the  $n$  coded inputs to a maximum of  $2^n$  unique outputs. We can have 2-to-4 decoder, 3-to-8 decoder, 4-to-16 decoder, etc. Figure 3.12 shows the truth table and logic diagram of a 2-to-4 decoder. The output lines are typically numbered with the decimal equivalent of the input binary code. In the truth table, a  $\times$  symbol stands for a *don't care condition*. When a Boolean function does not have any values specified for some input combinations, the input combinations are called don't care conditions. The truth table entry associated to a don't care condition is typically marked with a  $\times$ . Entries for input values can also be marked with a  $\times$ . The value of a don't care entry in the truth table does not matter and can be chosen as 1 or 0 depending on the situation.

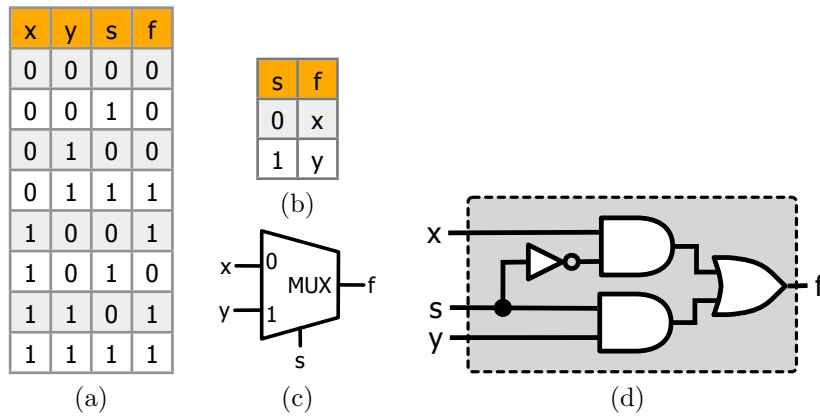


Figure 3.14: A 2-to-1 MUX: (a) its truth table; (b) its function table; (c) its graphic symbol; (d) its logic diagram.

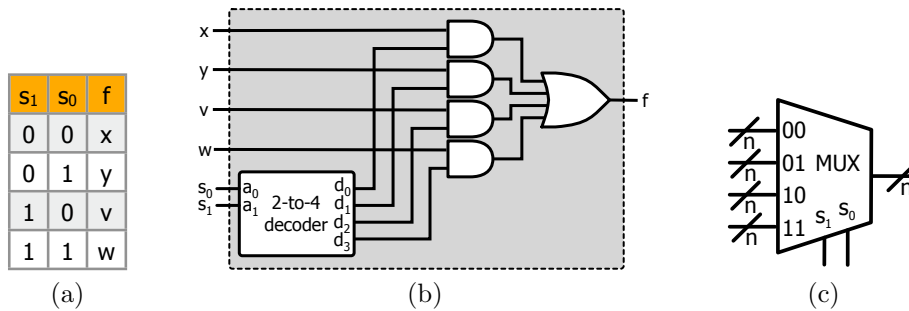


Figure 3.15: A 4-to-1 MUX: (a) its realization with a 2-to-4 decoder; (b)  $n$ -bit 4-to-1 MUX.

A decoder often contains a control input called *enable*. When the enable input is 1, the outputs of the decoder are enabled. Otherwise, all the outputs are 0. We can build a 3-to-8 decoder by combining two 2-to-4 decoders with an enable input. Its logic diagram is shown in Figure 3.13. The MSB ( $a_2$ ) of the input is used to enable the upper decoder when it is 0. When it is 1, the upper decoder is disabled and the lower decoder is enabled in the logic diagram.

## Multiplexers

A *multiplexer* (or *selector*) is a digital switch that connects data from one of  $2^n$  sources to its output. *MUX* is a shorthand for multiplexer. It has  $n$  selection bits,  $2^n$  input lines, and a single output. The selection bits select one of the  $2^n$  sources for its output. A 2-to-1 MUX is modeled with the following Boolean

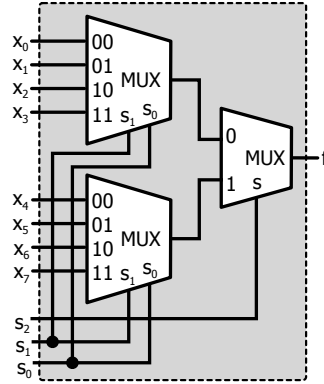


Figure 3.16: Building a 8-to-1 MUX using two 4-to-1 and one 2-to-1 MUXes.

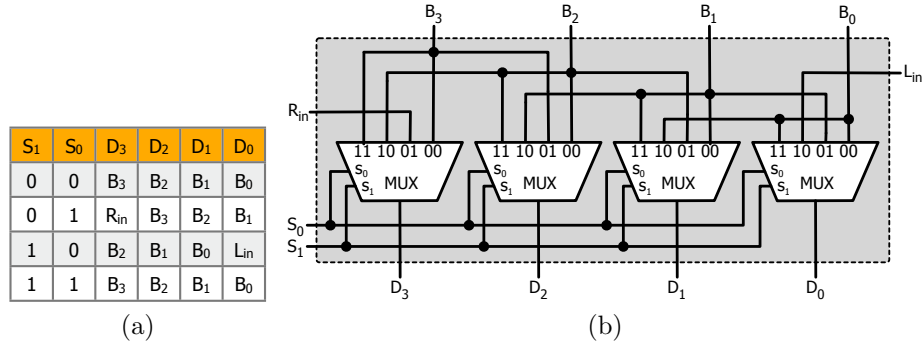


Figure 3.17: A 4-bit shifter: (a) its truth table; (b) its logic diagram.

function:

$$f(x, y, s) = (x \wedge s') \vee (y \wedge s)$$

where  $x$  and  $y$  are the two inputs, and  $s$  is the selection bit. Figure 3.14 (a) shows the truth table of a 2-to-1 MUX. This table can be further simplified to the function table shown in Figure 3.14 (b) with regards to the  $x$  and  $y$  inputs. It lists the input that is passed to the output for each value of the selection bit  $s$ . If  $s = 1$ , the input  $x$  is connected to the output. Otherwise, the input  $y$  is connected to the output. A *function table* is a condensed form of truth table. A single row in the function table represents many rows in the corresponding full truth table. Figure 3.14 (c) and (d) shows the graphic symbol and logic diagram of the 2-to-1 MUX.

The Boolean function for a 4-to-1 MUX is given by

$$f(x, y, v, w, s_1, s_0) = (x \wedge s'_0 \wedge s'_1) \vee (y \wedge s_0 \wedge s'_1) \vee (v \wedge s'_0 \wedge s_1) \vee (w \wedge s_0 \wedge s_1)$$

Figure 3.15 (a) shows the truth table for a 4-to-1 MUX. As shown in Figure 3.15 (b), a 4-to-1 MUX can be realized with a 2-to-4 decoder, four 2-bit AND gates,



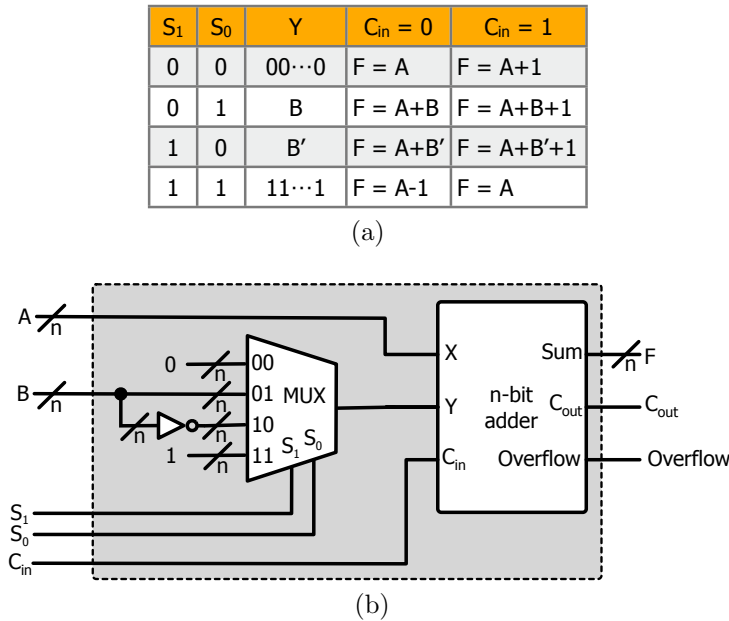


Figure 3.18: An  $n$ -bit arithmetic unit: (a) its function table; (b) its logic diagram.

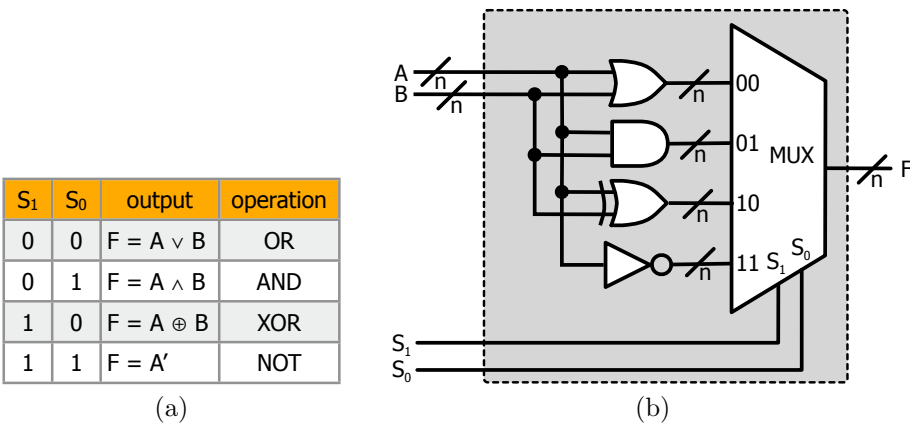


Figure 3.19: An  $n$ -bit logic unit: (a) its function table; (b) its logic diagram.

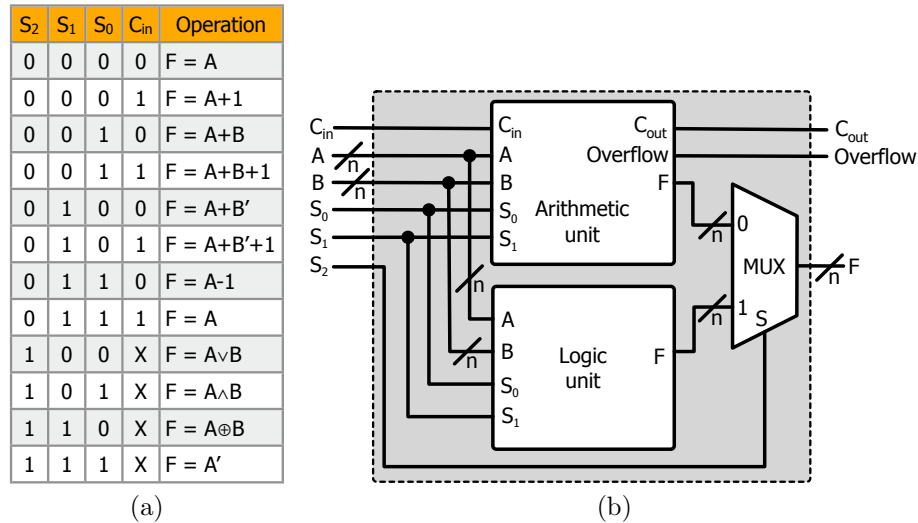


Figure 3.20: An  $n$ -bit ALU: (a) its function table; (b) its logic diagram.

and one 4-bit OR gate. Each output of the decoder is connected to an input of the MUX through an AND gate. The outputs of the four AND gates are combined together with an OR gate. Figure 3.15 (c) shows the graphic symbol of an  $n$ -bit 4-to-1 MUX. A larger MUX can also be constructed by combining smaller MUXs together. For example, as shown in Figure 3.16, an 8-to-1 MUX can be made with two 4-to-1 and one 2-to-1 MUXs. The two least significant selection bits connect to the two selection bits of 4-to-1 MUXs. Thus, a given input to the 8-to-1 MUX addresses a single output of each 4-to-1 MUX. The remaining selection bit  $s_2$  connects to the selection bit of the 2-to-1 MUX, which selects one of the two outputs from the two 4-to-1 MUXs.

### Shifters

Today's CPUs have at least some instructions for 1-bit left and right shift. As mentioned before, the shift operation can be performed in the logic or arithmetic mode. In the arithmetic mode, the word being shifted is assumed to be a two's complement binary number. Thus, the logical right shift and arithmetic right shift are handled differently. During the arithmetic right shift, the vacant bit positions due to the shift are filled in with the sign bit (MSB).

The shift instructions are implemented with a *shifter*. Figure 3.17 shows a realization of a 4-bit shifter using 2-to-4 MUXs. It has one 4-bit input  $B$ , two single bit inputs  $R_{in}$  and  $L_{in}$ , one 4-bit output  $D$ , and one 2-bit selection input  $S$ . This shifter moves the bits of a *nibble* (4 bits, half of a byte) one position to the left or right. When both of  $S_1$  and  $S_0$  are 0 or 1, the input is transferred unchanged to the output. When  $S_1 = 0$  and  $S_0 = 1$ , it performs a 1-bit right shift operation. It performs an arithmetic right shift operation if  $B_3$

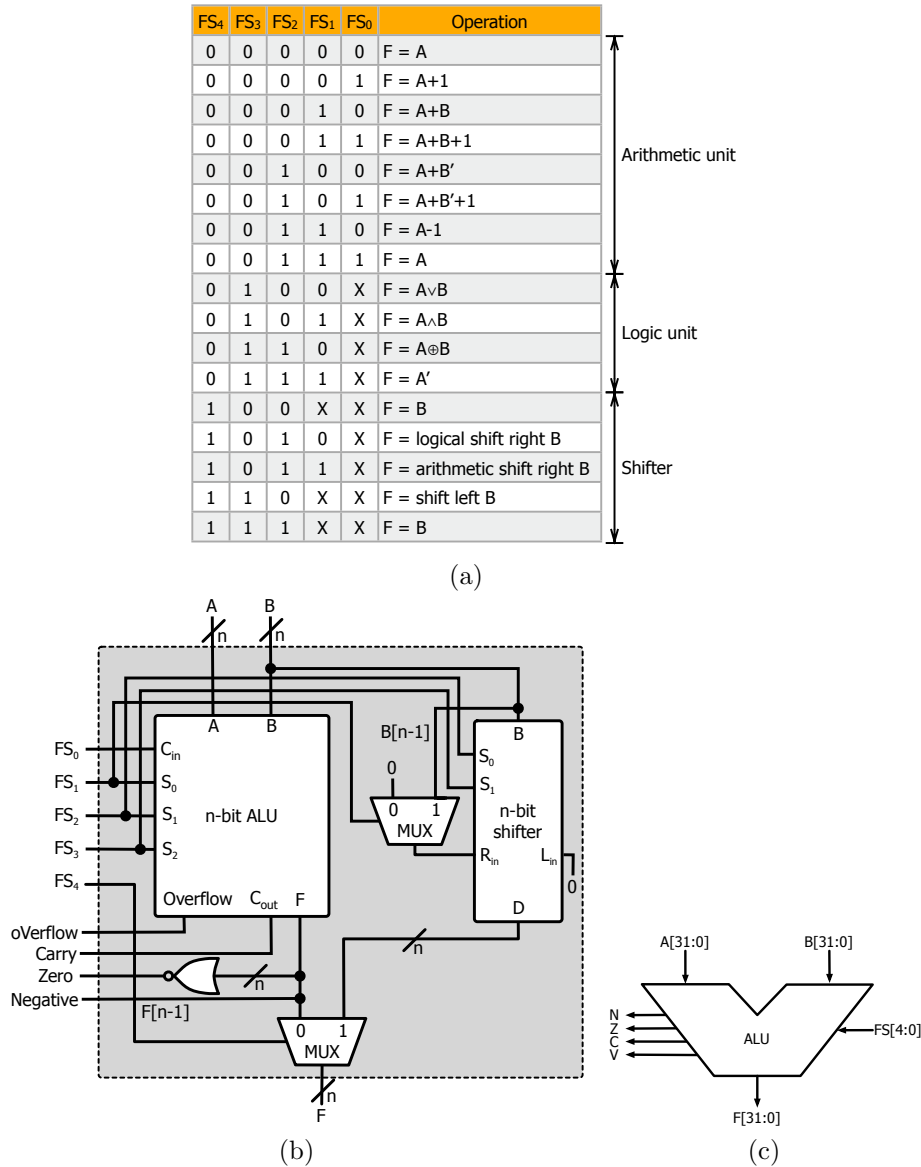


Figure 3.21: An  $n$ -bit ALU with a shifter: (a) its function table; (b) its logic diagram; (c) the graphic symbol of a 32-bit ALU.

is connected to  $R_{in}$ . If  $R_{in}$  is set to 0, it is a 1-bit logical right shift operation. When  $S_1 = 1$ ,  $S_0 = 0$ , and  $L_{in} = 0$ , it performs a 1-bit arithmetic or logical left shift operation.

To perform a shift operation of more than one position, the data needs to go through the shifter several times. To achieve high performance, other shifters,

such as a *barrel shifter*, is used. The barrel shifter shifts any number of positions at once.

### Arithmetic logic units

An arithmetic logic unit (ALU) is a logic circuit that performs arithmetic and logic operations and is a fundamental building block of the CPU. We can design an ALU to perform any complex operation. A simple  $n$ -bit ALU typically realizes the following operations:

- $n$ -bit integer arithmetic operations, such as  $n$ -bit binary addition and subtraction.
- $n$ -bit logic operations, such as  $n$ -bit AND, OR, NOT, and XOR.
- 1-bit shift operations, such as 1-bit arithmetic shift and 1-bit logical shift.

An  $n$ -bit arithmetic unit inside the ALU performs the integer addition and subtraction. Figure 3.18 shows a realization of the  $n$ -bit arithmetic unit using an  $n$ -bit binary adder. The  $n$ -bit inputs  $A$  and  $B$  are the operands of the operation specified by the function selection bits  $S_1$  and  $S_0$ . In the function table shown in Figure 3.18 (a),  $B'$  stands for the result after applying a bit-wise negation operation to  $n$ -bit binary number  $B$ . When  $S_1 = 1$ ,  $S_0 = 0$ , and  $C_{in} = 1$ , it performs an  $n$ -bit subtraction of two operands  $A$  and  $B$ . Since the two's complement representation of  $B$  is  $B' + 1$ ,  $A - B$  can be computed by performing  $A + B' + 1$ . Note that most of modern CPUs use the two's complement representation for their internal representation of numbers.

An  $n$ -bit logic unit inside the ALU performs  $n$ -bit AND, OR, NOT, and XOR operations for its operands  $A$  and  $B$ . It can be easily implemented with elementary logic gates and a MUX. Its function table and logic diagram are shown in Figure 3.19. After combining the arithmetic and logic units with an  $n$ -bit 2-to-1 MUX, we have a simple  $n$ -bit ALU as shown in Figure 3.20.

To make the ALU to perform 1-bit shift operations, we combine the ALU with an  $n$ -bit shifter (a 4-bit version is described in Figure 3.17). The final  $n$ -bit ALU is shown in Figure 3.21. The idea is the same as that of combining the arithmetic and logic units. However, to distinguish the arithmetic right shift from the logical right shift, we add a 2-to-1 MUX. The function selection input  $FS_1$  connects to the selection input of the MUX. When  $FS_1 = 1$ , the MSB of the operand  $B$  connects to  $R_{in}$  of the shifter, and it performs a 1-bit arithmetic right shift. When  $FS_1 = 0$ ,  $R_{in}$  is set to 0, and the shifter performs a 1-bit logical right shift. The graphic symbol shown in Figure 3.21 (c) stands for a 32-bit ALU. Its inputs are two 32-bit operands  $A$  and  $B$ , and five ALU function selection bits  $FS[4 : 0]$ .  $F$  is the result of the ALU operation.  $N$ ,  $Z$ ,  $C$ , and  $V$  are flags that show the status of the computation performed in the ALU.  $N$  and  $Z$  indicate that the result is negative and zero, respectively.  $C$  is the carry from the MSB during the ALU operation. Finally,  $V$  indicates that an overflow occurred during the operation. The control unit in the CPU uses these flags to implement *branches*.

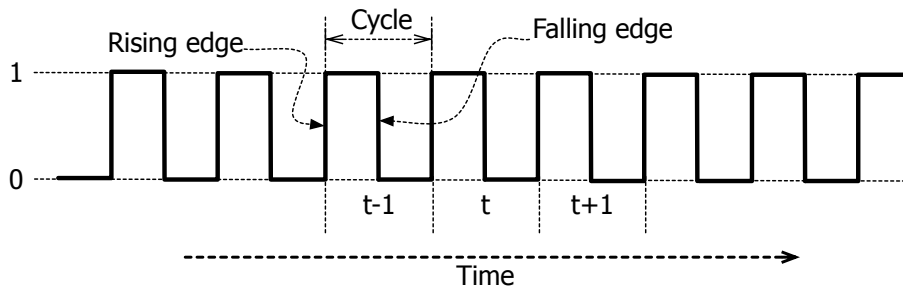


Figure 3.22: Clock signals.

### 3.4 Sequential Logic Circuits

The outputs of a sequential logic circuit are functions of the current input values and the data stored in memory (i.e., the internal state). Almost all sequential logic circuits today employ a *clock* to precisely control the times at which data is transferred to or from memory elements. They are said to be *clocked* or *synchronous*. Thus, the outputs of a sequential logic circuit are essentially a function of time.

A clock delivers a continuous train of alternating signals (*pulses*) as shown in Figure 3.22. It oscillates between 0 and 1. The *clock signal* is simultaneously broadcast to every circuit component. A *clock cycle* is a fixed interval of time between two clock pulses. Each clock cycle models one discrete time unit. For example, a 1GHz clock has 1 nano-second clock cycle time. Circuit components may become active at either the *rising edge* (also known as the *positive edge*), *falling edge* (also known as the *negative edge*), or both of the rising and falling edges of a clock cycle. Without loss of generality, we assume each circuit component is active at the positive (rising) edge throughout this book.

The basic memory element in a sequential logic circuit is a *flip-flop*. It has two stable states and capable of storing one bit of information. There are many different types of flip-flops. Among those, we introduce *D flip-flop* in this chapter.

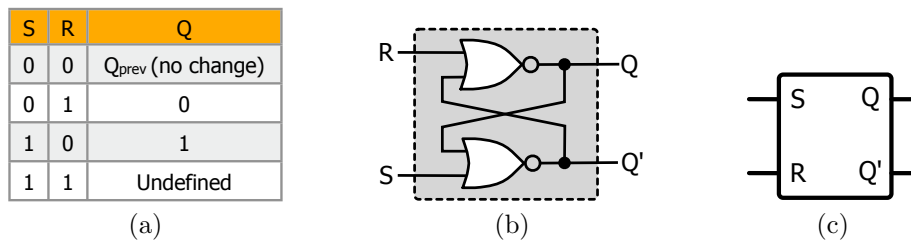


Figure 3.23: An SR latch: (a) its function table; (b) its logic diagram; (c) its graphic symbol.

## Latches

Flip-flops are typically made from *latches*. They are also bistable memory elements that have two distinct stable states. While the output of a flip-flop changes only with respect to the clock signal, a latch changes its outputs whenever its inputs change. The *SR latch* is one of the simplest memory elements. Its function table, logic diagram, and graphic symbol are shown in Figure 3.23. It has two outputs  $Q$  and  $Q'$ , and has two stable states:  $Q = 0$  (*reset*) and  $Q = 1$  (*set*).  $Q'$  is the complement of  $Q$  except when both  $S$  and  $R$  set to 1.  $Q'$  becomes the complement of  $Q$  only after the state of the latch has stabilized. The function table is similar to those used for combinational circuits. However, the function table for the SR latch implies a time dependence. For example, in the first row of the function table, the value  $Q_{prev}$  of the output  $Q$  refers to the output (i.e., state) prior to applying  $S = 0$  and  $R = 0$ . When  $S = 0$  and  $R = 0$ , the SR latch does not change its state. Since it remains in a particular state with these inputs, it is named a latch.  $S = 1$  and  $R = 0$  *sets* the latch by changing its state to  $Q = 1$ .  $S = 0$  and  $R = 1$  *resets* the latch by changing its state to  $Q = 0$ . When  $S = 1$  and  $R = 1$ , the resulting state is undefined. The behavior of the SR latch is described by the following equation:

$$Q = S \vee (R' \wedge Q_{prev})$$

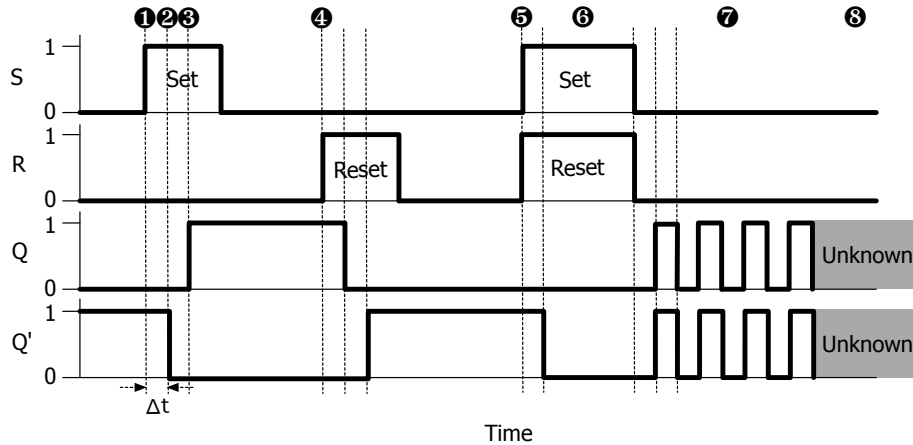


Figure 3.24: A timing diagram of the SR latch.

Figure 3.24 shows a *timing diagram* of the SR latch. A timing diagram shows how a set of digital signals vary over time. Assume that both inputs are 0 and the output  $Q$  is 0 (thus  $Q' = 1$ ) at the beginning. In addition, let the gate delay of a NOR gate in the SR latch be  $\Delta t$ . Setting the latch (❶ in Figure 3.24) causes  $Q'$  to go to the state 0 after the propagation delay  $\Delta t$  of a single NOR gate (❷ in Figure 3.24).  $Q$  goes to the state 1 after the propagation delay  $2 \cdot \Delta t$  of two NOR gates (❸ in Figure 3.24). The analysis of the timing diagram

when resetting the latch is similar to the case of setting it (④ in Figure 3.24). Attempting to set and reset the latch simultaneously (⑤ in Figure 3.24) causes both  $Q$  and  $Q'$  to be 0 (⑥ in Figure 3.24). Making the  $S$  and  $R$  inputs back to 0 may cause both  $Q$  and  $Q'$  to oscillate (⑦ in Figure 3.24) until the SR latch enters an unknown stable state (⑧ in Figure 3.24).

The SR latch can be used to store a single bit of information. As long as the two inputs  $S$  and  $R$  are 0, the output  $Q$  remains unchanged. To store 1,  $S$  is set to 1, and then it is set back to 0 again. This makes the output  $Q$  changed to 1 and then remain unchanged. To store 0, we set  $R$  to 1 and set it back to 0.

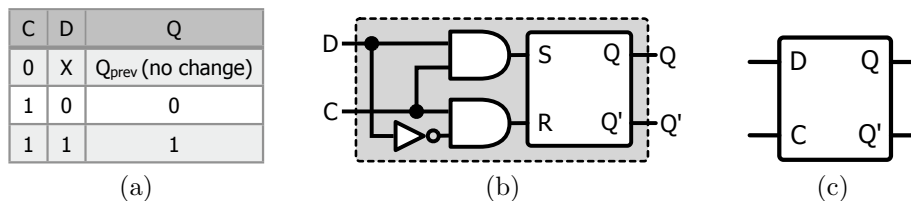


Figure 3.25: A D latch: (a) its function table; (b) its logic diagram; (c) its graphic symbol.

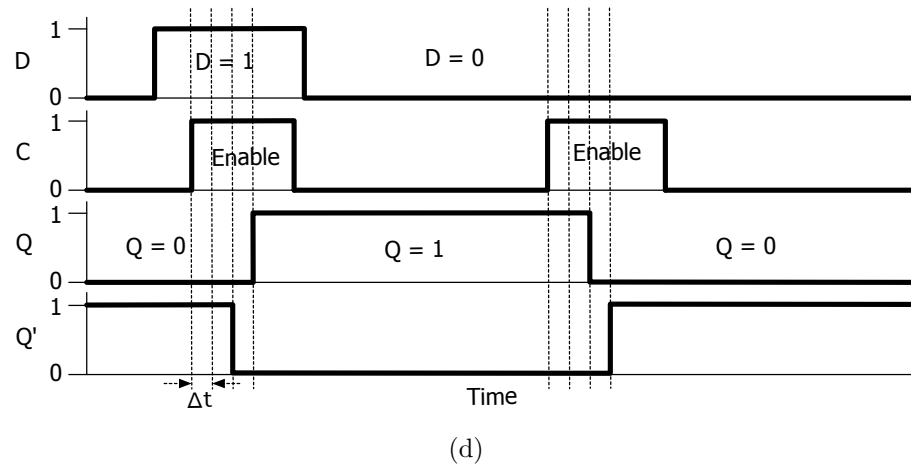


Figure 3.26: Timing diagram of the D latch.

Unlike SR latches, a *D latch* does not have any undefined state. Figure 3.25 (a) shows the function table of the D latch. It has two inputs:  $D$  and  $C$ .  $D$  is a data input and specifies the next value to be stored in it.  $C$  is a control input (called enable) and controls receiving a new value or keeping the old value already stored. When setting  $C$  to 1 (i.e., enabled),  $Q$  becomes the same as the input  $D$ . Otherwise,  $Q$  remains unchanged. The D latch implementation in Figure 3.25 (b) uses an SR latch. Figure 3.25 (d) shows a timing diagram of

the D latch (we assume that the gate delay of the NOT gate is negligible in this timing diagram).

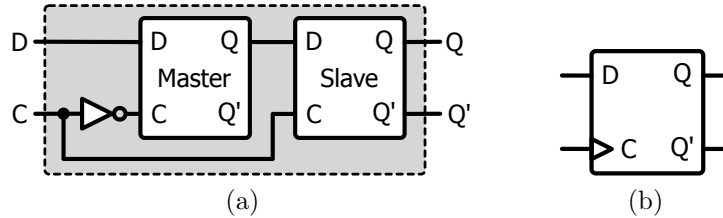


Figure 3.27: A D flip-flop: (a) its logic diagram; (b) its graphic symbol.

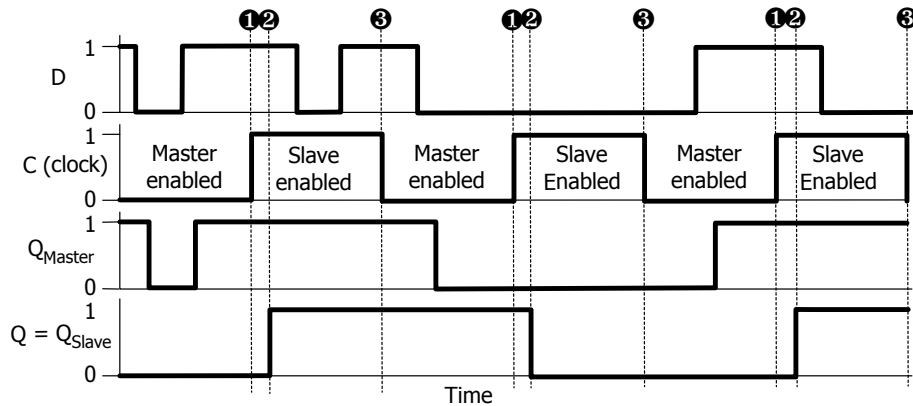


Figure 3.28: A timing diagram for a positive edge-triggered D flip-flop.

### The D flip-flop

As long as the control input  $C$  remains 1, the output of the D latch will momentarily change each time its input changes. Because of this property, the D latch is called *transparent*. Assume that we connect a clock signal to the control input of a D latch, say  $x$ , and its output connects to the inputs of other D latches that share the same clock signal for synchronization. Then, any change in the input of the D latch  $x$  makes the outputs of other D latches changed while the clock signal is 1. Thus, some additional changes in the input of the D-latch  $x$  may cause an undesirable effect on the entire system. Unlike latches, flip-flops are not transparent. This is the key difference between latches and flip-flops.

Figure 3.27 (a) shows how to build a D flip-flop using two D latches. The D flip-flop captures its input  $D$  at the rising edge or falling edge of the clock cycle. If it samples its input on the rising edge, it is called a *positive edge-triggered flip-flop*. If it samples its input on the falling edge, it is called a *negative*



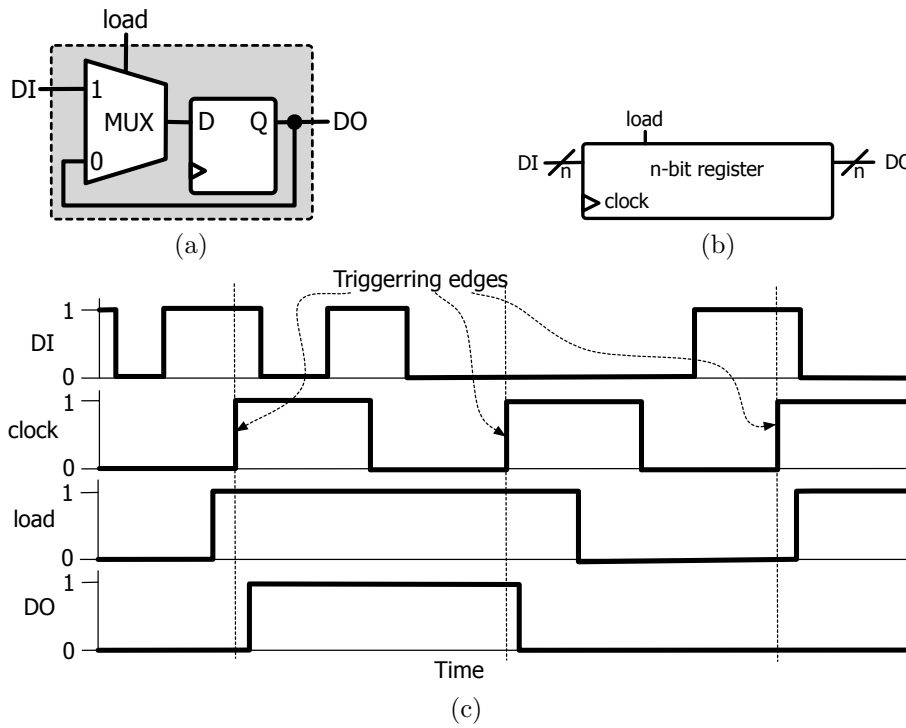


Figure 3.29: Registers: (a) the logic diagram of a single-bit register; (b) an  $n$ -bit register; (c) a timing diagram for a positive edge-triggered single-bit register.

*edge-triggered flip-flop.* As mentioned before, we assume that we use positive edge-triggered flip-flops throughout the book. The captured value becomes the output  $Q$  of the D flip-flop. At other times  $Q$  remains unchanged. The realization of a D flip-flop shown in Figure 3.27 (a) is known as a *master-slave* circuit. The negation of the clock signal is connected to the clock input of the master D latch, while the clock signal is directly connected to the control input of the slave D latch.

Figure 3.28 shows a timing diagram for the positive edge-triggered D flip-flop. While the external clock signal input  $C$  is 0, the master latch is enabled and the slave latch is disabled. The master keeps capturing the data on the input  $D$ . While  $C$  is 1, the master latch is disabled and the slave latch is enabled. The output  $Q$  of the master is transferred to the input of the slave. The slave keeps capturing the data on its input.

When the external clock signal goes from 0 to 1 (the rising edge, ❶ in Figure 3.28), the master becomes disabled and its output remains unchanged. The slave becomes enabled. The data that is captured on the rising edge by the master is transferred to the slave, and the data appears on the output of the D flip-flop (❷ in Figure 3.28).

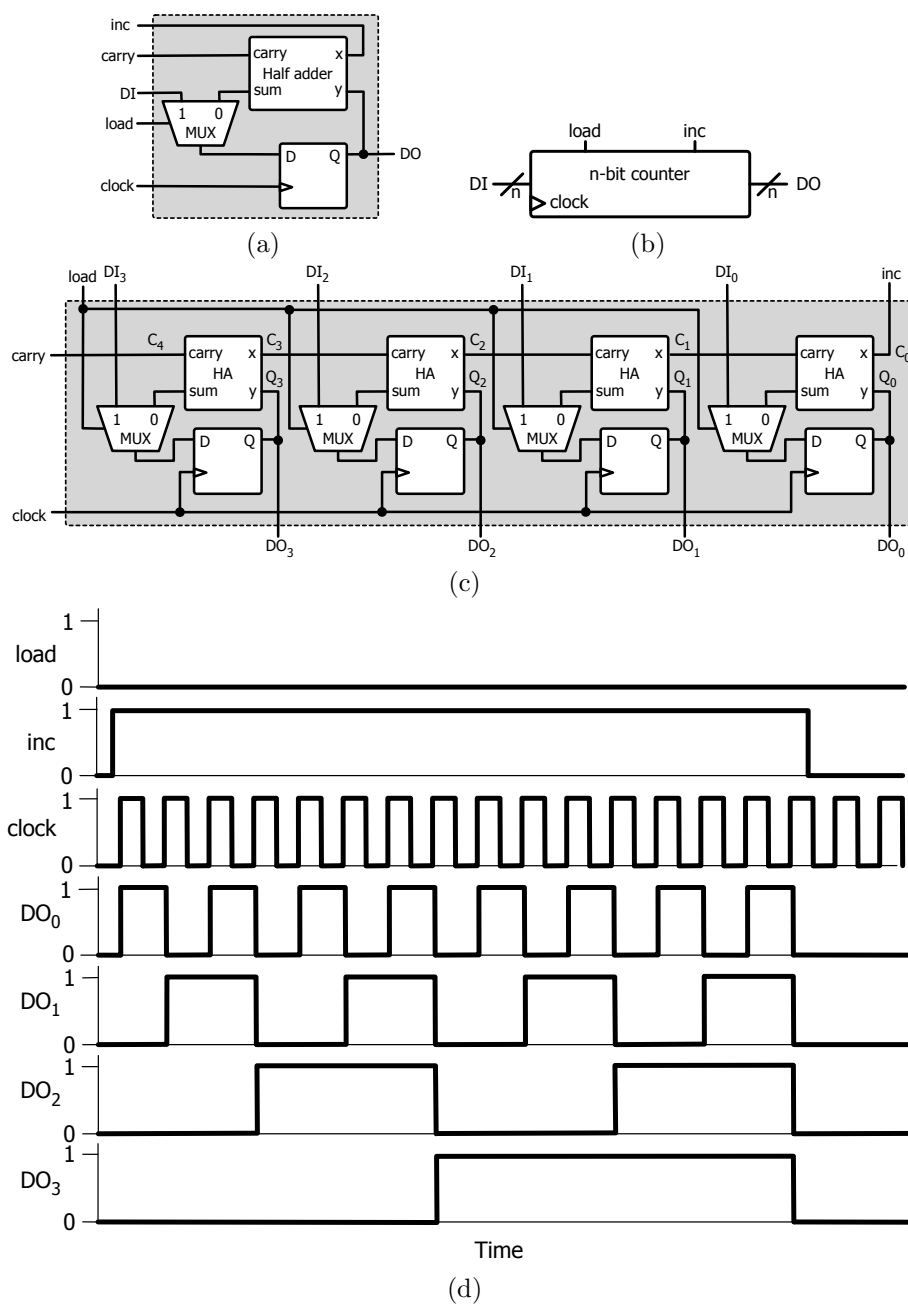


Figure 3.30: Counters: (a) the logic diagram of a single-bit counter; (b) the graphic symbol of the  $n$ -bit counter; (c) the logic diagram of a 4-bit counter; (d) a timing diagram for the 4-bit counter.

When the external clock signal goes from 1 to 0 (the falling edge, ❸ in Figure 3.28), the master is enabled and starts to capture data again. The slave becomes disabled and its output remains unchanged. Since the external clock signal reaches the slave faster than the master because of the gate delay of the NOT gate, the slave is guaranteed to be disabled before any changes of the master's input go through the master and reach the slave's input.

Flip-flops sometimes provide special inputs for setting or clearing them asynchronously without the need for a clock pulse. These inputs are called *preset* and *clear*. Preset sets the output of the flip-flop to 1 and clear sets it to 0. They are useful for setting up flip-flops to an initial state after power is turned on.

### Registers

A *register* is a storage device that can store binary information over time. It is a collection of one or more flip-flops. Figure 3.29 (a) shows the logic diagram of a single-bit register that consists of one D flip-flop and one 2-to-1 MUX. It has three input lines *load*, *clock*, and *DI* and one output line *DO*. To store a value, we put the value in the input *DI* and activate *load* (i.e., setting it to 1). In the next clock cycle (i.e., on the positive edge of the next clock cycle), the data on the input *DI* is transferred into the register. When *load* = 0, the output *Q* of the D flip-flop is connected to its own input *D*. On the next positive edge of the clock signal, the D flip-flop captures the value of *Q*, and its state remains unchanged.

An *n*-bit register can be realized with an array of *n* single-bit registers. The symbol for an *n*-bit register is shown in Figure 3.29 (b). The multi-bit content of such a register is typically referred to as a *word*.

Figure 3.29 (c) shows a timing diagram of the single-bit register. Each positive edge of the clock signal, the register accepts the new data value on its input *DI*.

### Binary counters

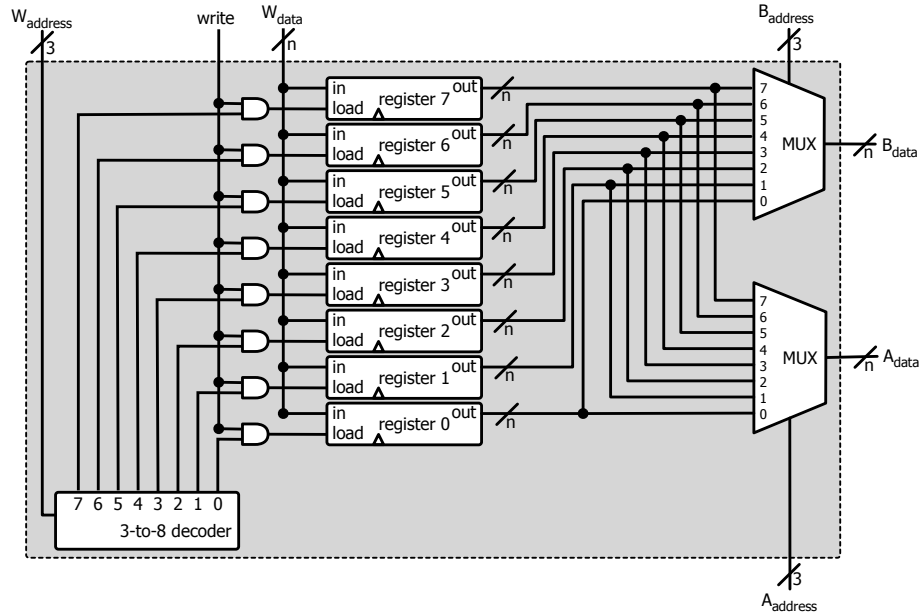
An *n*-bit *counter* is an *n*-bit register that goes through a predetermined sequence of states when the clock signal is applied. An *n*-bit *binary counter* determines its *n*-bit output *DO* in the next clock cycle in the following way:

$$DO(t+1) = (DO(t) + 1) \bmod 2^n$$

where  $x \bmod m$  for two integers  $x$  and  $m$  is defined by

$$x \bmod m = x - m \left\lfloor \frac{x}{m} \right\rfloor$$

Figure 3.30 (a) shows the logic diagram of a 1-bit binary counter that is designed with a D flip-flop, a 2-to-1 MUX, and a half adder. It has an increment input *inc*. When *inc* = 1 and *load* = 0, it increments its output in every clock cycle. That is, it adds *inc* and the current output *DO*. The result appears on the output *DO* in the next clock cycle. Similar to a register, the input *load*

Figure 3.31: Logic diagram of an  $n$ -bit register file.

makes the counter take the new value on the input  $DI$ . This value appears on  $DO$  in the next clock cycle.

Figure 3.30 (b) shows the graphic symbol of an  $n$ -bit counter. Combining  $n$  1-bit counters makes an  $n$ -bit counter. Figure 3.30 (c) shows the logic diagram of a 4-bit counter. The half adder in the  $i^{th}$  bit position adds  $Q_i$  from the  $i^{th}$  D flip-flop and  $C_i$  from the half adder in the  $(i - 1)^{th}$  bit position. The input  $inc$  to the 4-bit counter connects to one of the two inputs of the half adder in the LSB position.

Figure 3.30 (d) shows a timing diagram of the 4-bit counter. At the beginning,  $DO = 0000_2$ . When  $inc = 1$ , the counter starts counting the clock pulses. The output  $DO$  follows a binary number sequence: 0000, 0001, 0010,  $\dots$ , 1111, 0000. When  $inc$  becomes 0, the counter stops counting. As long as  $inc = 1$  and  $load = 0$ , this counter adds 1 to its content, say  $X$ , on every positive edge of the clock signal. The content becomes  $(X + 1) \bmod 16$  from then on.

### Register files

A *register file* is an array of registers in the CPU. It consists of  $2^m$  rows of  $n$  flip-flops. It can be thought as  $2^m$  rows of  $n$ -bit registers. Figure 3.31 shows an implementation of a register file that contains  $2^3$   $n$ -bit registers. When  $write = 1$ , the 3-bit address input  $W_{address}$  to the 3-to-8 decoder selects a register where the value on the data input  $W_{data}$  will be stored. It happens on the next positive edge of the clock signal. When  $write = 0$ , the value on the

data input  $W_{data}$  is ignored. Using the 3-bit address input  $A_{address}$ , the register file selects a register whose content appears on the data output  $A_{data}$ . Similarly,  $B_{address}$  selects a register for  $B_{data}$ .  $A_{address}$  and  $B_{address}$  are possibly the same.  $W_{data}$  is called a *write port*.  $A_{data}$  or  $B_{data}$  is called a *read port*.

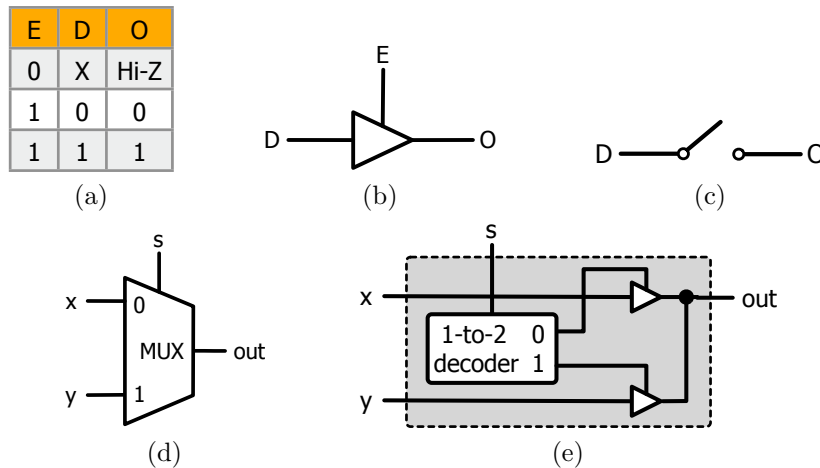


Figure 3.32: The tristate buffer: (a) its function table; (b) its graphic symbol; (c) when  $E = 0$ , it is treated as an open switch; (d) the graphic symbol of a 2-to-1 MUX; (e) a logic diagram that realizes the 2-to-1 MUX using tristate buffers and a 1-to-2 decoder.

### Tristate buffers

The output of a *tristate buffer* (or *tristate driver*) allows a third state, called a *high-impedance state* and denoted as  $Hi - Z$ , in addition to 0 and 1. The function table and graphic symbol for a tristate buffer is given in Figure 3.32. When  $E = 1$ , the output  $O$  just follows the input  $D$ . When  $E = 0$ , the output  $O$  is in the high-impedance state ( $Hi - Z$ ) regardless of the input. A tristate buffer can be thought of as a switch. When its output is in the high-impedance state, it is treated as an open switch.

Using tristate buffers, multiple circuits may share the same wire. For example, with  $2^m$  tristate buffers and a  $m$ -to- $2^m$  decoder, one can implement a  $2^m$ -to-1 MUX. Figure 3.32 (e) shows a logic diagram that realizes a 2-to-1 MUX in Figure 3.32 (d) using two tristate buffers and a single 1-to-2 decoder. According to the value of the selection bit  $s$ , the circuit connects one of its inputs ( $x$  or  $y$ ) to the output line.

### Random access memories

Binary information is stored in memory as groups of  $w$  bits. Each group of  $w$  bits is called a word. Random access memory (RAM) is one of the major

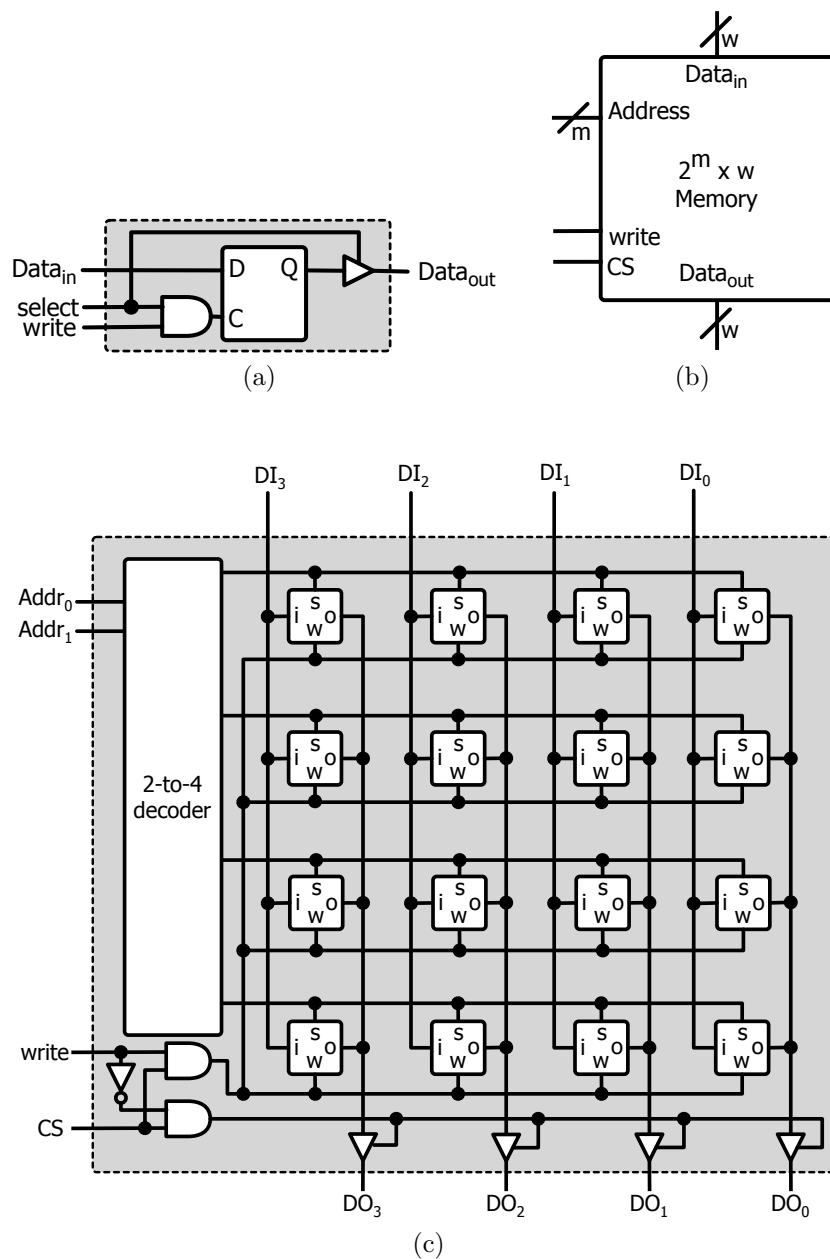


Figure 3.33: RAM: (a) the logic diagram of a RAM cell; (b) the graphic symbol of a  $2^m \times w$  RAM; (c) the logic diagram of a  $4 \times 4$  RAM.

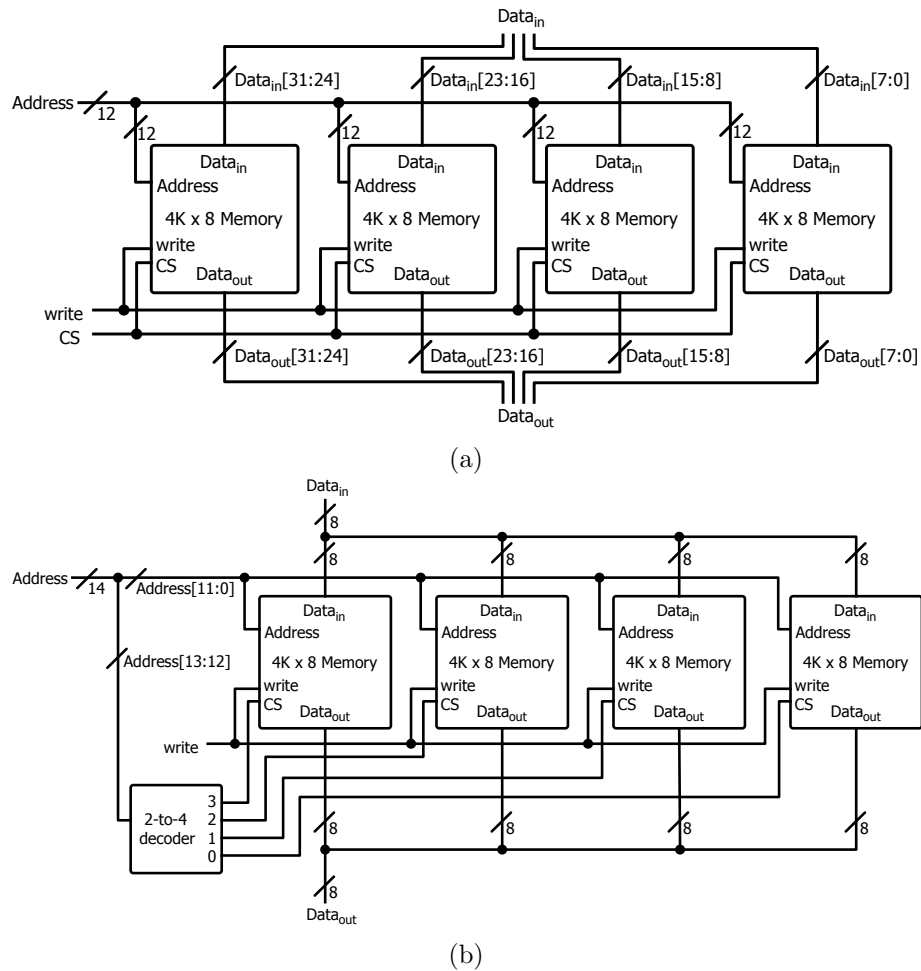


Figure 3.34: A larger memory can be constructed from smaller RAM chips: (a) constructing a  $4K \times 32$  RAM with four  $4K \times 8$  RAMs; (b) constructing a  $16K \times 8$  RAM with four  $4K \times 8$  RAM.

components of the von Neumann architecture. It is able to access randomly chosen words in the RAM regardless of the order in which they are accessed. The capacity of a memory unit is typically described as the total number of bytes that can be stored in it. In addition, the configuration of a RAM is typically described as follows:

$$(\text{the number of words}) \times (\text{the number of bits in a word})$$

The symbol K (kilo), M (mega), and G (giga) typically stand for factors of  $10^3$ ,  $10^6$ , and  $10^9$  respectively. However, when refer to the capacity of memory, they stand for  $2^{10}$ ,  $2^{20}$ , and  $2^{30}$  respectively. For example,  $64K \times 8$  RAM can store

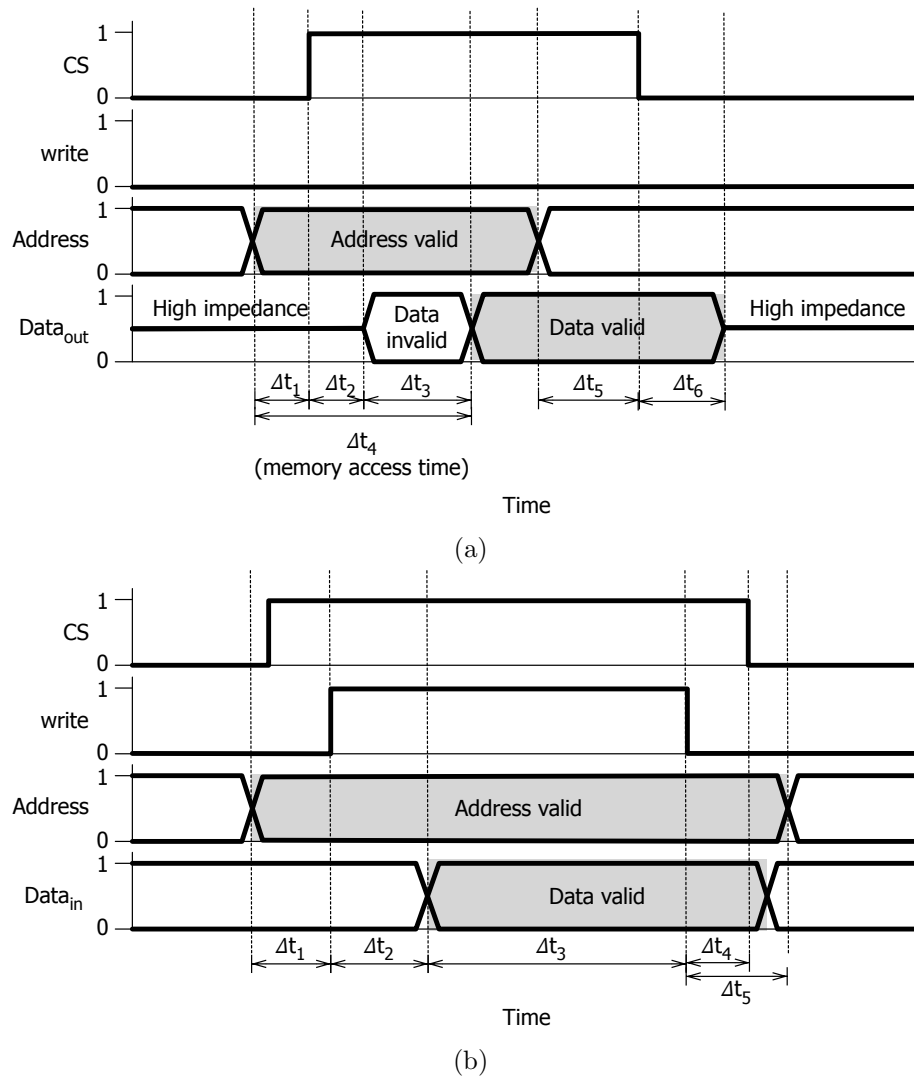


Figure 3.35: Read and write cycles of a RAM: (a) read cycle; (b) write cycle.

total  $64 \times 2^{10} \times 8 \div 8 = 65,536$  bytes.

A RAM is organized as a matrix of memory cells. The logic diagram of a memory cell is given in Figure 3.33 (a). It is a combination of a D latch, an AND gate, and a tristate buffer. When the selection input *select* that controls the tristate buffer is 1, the bit stored in the D latch will be on the data output *Data<sub>out</sub>*. If *select* = 1 and *write* = 1, the value on the data input *Data<sub>in</sub>* will be stored in the latch.

For example, arranging 16 memory cells with a 2-to-4 decoder, we can build



a  $4 \times 4$  RAM as shown in Figure 3.33 (c). Since there are total  $4 = 2^2$  different words in the RAM, two address bits  $Addr_1$  and  $Addr_0$  are enough to address all the words. The tristate buffer in each memory cell enables sharing the data input line and output line between different memory cells. A RAM chip typically provides a chip select input,  $CS$ . The  $CS$  input is similar to an on-off switch. When  $CS = 0$ , all data output lines ( $DO$ ) are in the high-impedance state. The data output lines are effectively disconnected from the outside of the RAM chip. Neither a read operation nor a write operation can be performed. When  $CS = 1$  and a 2-bit address is presented to the address lines  $Addr$ , the control input  $write$  selects between the read operation ( $write = 0$ ) and write operation ( $write = 1$ ).

Figure 3.33 (b) shows a graphical symbol for a  $2^m \times w$  RAM. It has  $w$  data input lines,  $w$  data output lines,  $m$  address lines, a  $CS$  line, and a  $write$  line. Each word in the RAM has a unique address ranging from 0 to  $2^m - 1$ .

Using the  $CS$  input provided by each RAM chip, a memory with bigger capacity can be made up of several RAM chips with smaller capacity. For example, Figure 3.34 (a) shows the design of a  $4K \times 32$  RAM using four  $4K \times 8$  RAM chips, and Figure 3.34 (b) shows how to construct a  $16K \times 8$  RAM with four  $4K \times 8$  RAM chips and a 2-to-4 decoder.

The read and write operations of the RAM is typically controlled by the control unit in the CPU. The CU synchronizes it with other components. Due to the propagation delay, there are some timing requirements for accessing a RAM chip. Timing parameters of a RAM chip can be found in the datasheet provided by its vendor. Consider the timing diagram for a memory read cycle shown in Figure 3.35 (a). The points where the two lines crossing each other in the diagram represent changes of the values on multiple lines. Note that  $Address$  and  $Data_{out}$  are multi-bit binary values.

First, the address for the desired location is presented to the address input lines  $Address$ . Since the delay path from the address input to the output is longer than any other path from an input to the output, the address input must be presented before any other signals. After  $\Delta t_1$  time has been passed,  $CS$  becomes 1. It takes  $\Delta t_2$  time for the output lines to switch from the high-impedance state to the state 0 or 1. Some additional  $\Delta t_3$  time is required before the valid data appears on the output lines. The maximum time ( $\Delta t_1 + \Delta t_2 + \Delta t_3 = \Delta t_4$ ) taken between the application of the address to the address lines and the appearance of the valid data on the data output lines is called *memory access time*. After the address has been changed, the valid data is still available on the output lines for  $\Delta t_5 + \Delta t_6$  time. Since it takes time  $\Delta t_6$  for the output lines to enter the high-impedance state, the valid data is still on the output lines after  $CS$  is set to 0.

Figure 3.35 (b) shows the timing diagram for a memory write cycle. When writing to the RAM, it must be ensured that the address of the desired location is present before  $write$  is set to 1 ( $\Delta t_1$ ). If not, the incoming data may overwrite the existing data in a different location. The valid data must be stable for  $\Delta t_3$  time prior to setting  $write$  back to 0. Since  $write$  is connected to the  $C$  input of the D latch in each memory cell, the data to be stored in the D latch is the

data sampled on the falling edge of the *write* signal. This data must be stable for  $\Delta t_3 + \Delta t_4$  time before and after the falling edge of the *write* signal to ensure proper operation of the D latch. In addition, the valid data and valid address must be stable for  $\Delta t_4$  time and  $\Delta t_5$  time, respectively after *write* goes to 0 to avoid overwriting other locations.

## CHAPTER 4

### A Simple Computer Architecture

Modern computers are *digital systems* that process digital information. The digital information is represented by two discrete values 0 and 1. A digital system is typically divided into two parts: a datapath and a control unit. As a matter of fact, most CPUs consist of these two parts. The datapath is a collection of functional units, registers, and interconnections between them that together perform data-processing operations.

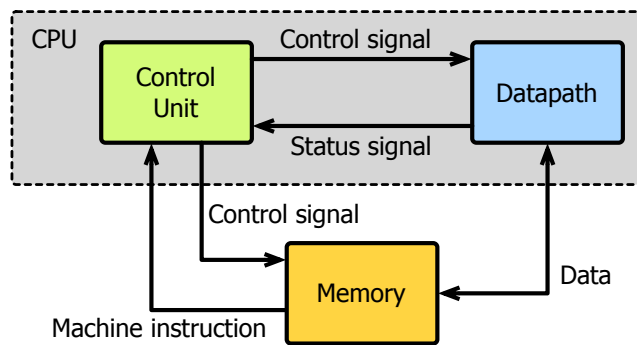


Figure 4.1: The relationship between the control unit, datapath, and main memory.

The *control unit* (CU) controls operations of the datapath and determines the sequence of the operations. It also coordinates interactions between the datapath and main memory (Figure 4.1). Machine instructions in a program stored in main memory are fetched one by one. Each machine instruction is decoded by the CU, and the CU generates control signals required to execute the instruction. The datapath sends status signals to the CU after executing an instruction. The CU takes an appropriate action according to the status signal.

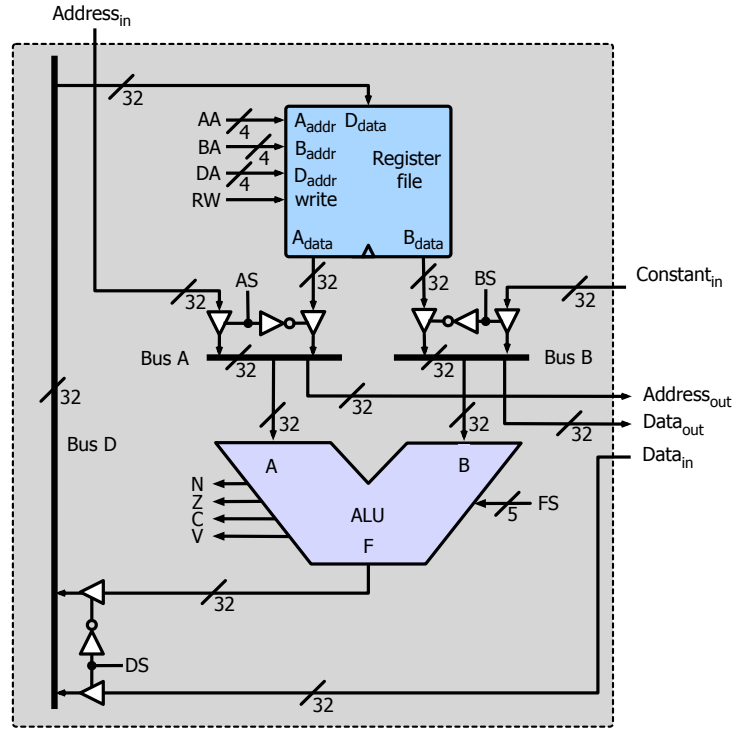


Figure 4.2: A 32-bit datapath.

### 4.1 Datapath

Figure 4.2 shows a simple 32-bit datapath that consists of a register file and an ALU. The organizations of the register file and ALU are explained in the previous chapter. The input  $AA$  ( $BA$ ) to the register file selects a register for the output  $A_{data}$  ( $B_{data}$ ) that is connected to the bus A (B).

The selection input  $AS$  connects either  $A_{data}$  from the register file or the address  $Address_{in}$  from outside the datapath to the bus A. Similarly, the selection input  $BS$  connects either  $B_{data}$  or the data  $Constant_{in}$  from outside the datapath to the bus B. The operands A and B of the ALU come from the bus A and bus B, respectively. The bus A also connects to  $Address_{out}$  to send an address outside the datapath to other components of the system (e.g., memory). Similarly, the bus B connects to  $Data_{out}$  to send data outside the datapath (e.g., to memory). An ALU operation is selected by the 5-bit input  $FS$  to the ALU. The selection input  $DS$  for the bus D selects one between the output  $F$  of the ALU and the data  $Data_{in}$  from outside of the datapath. Specifying a register using the input  $DA$  and activating the input  $RW$  make the value appearing on the bus D to be transferred to the specified register.

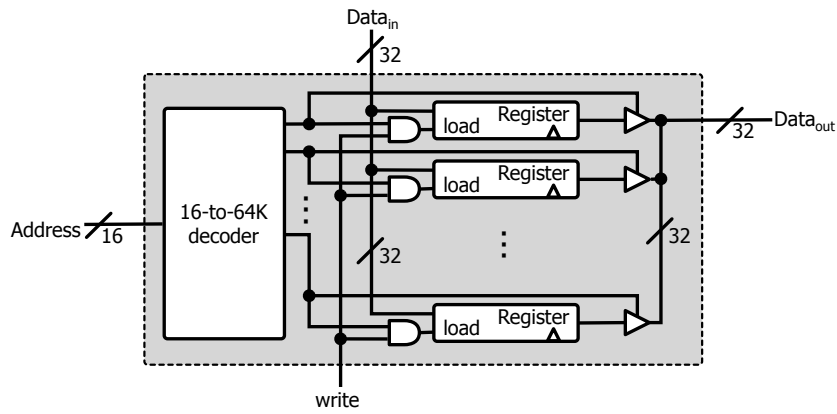


Figure 4.3: The memory treated as an array of registers.

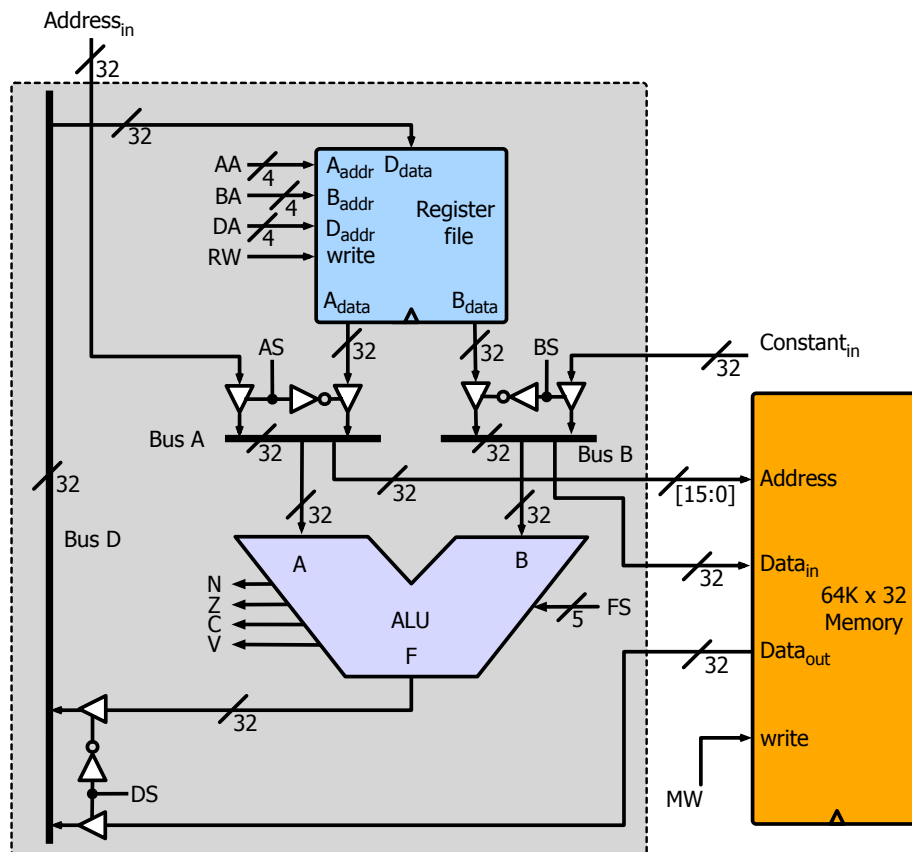
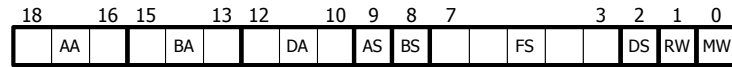


Figure 4.4: Connecting a memory unit to the datapath.



(a)

Control bits	Description
AA	specifies a register for the value on the bus A
BA	specifies a register for the value on the bus B
DA	specifies a register to which the value on the bus D is loaded
AS	selects one between the value of a register and $Address_{in}$ for the value on the bus A
BS	selects one between the value of a register and $Constant_{in}$ for the value on the bus B
FS	selects an ALU operation
DS	selects one between the result of the ALU and the data from the memory for the value on the bus D
RW	stores the value on the bus D to the register specified by DA
MW	stores the value that appears on the bus B into the memory location specified by the address on the bus A

(b)

Figure 4.5: The control word: (a) its format; (b) the description of its content.

## 4.2 Attaching a RAM to the Datapath

The steps required to read or write a word in the memory can be precisely controlled by the CU to satisfy the timing requirements. However, the easiest way of ignoring the details of the timing requirements is treating the memory as an array of  $2^m$   $w$ -bit registers as shown in Figure 4.3, where  $m$  is the number of address bits and  $w$  is the word size. In this case, storing a value into the memory occurs on every positive edge of the clock signal when  $MW$  is 1. Such a memory can be attached to the datapath in the way as shown in Figure 4.4. The address output  $Address_{out}$  from the bus A in the datapath is connected to the address input lines of the memory. The data output  $Data_{out}$  from and the data input  $Data_{in}$  to the datapath are connected to the data input lines and data output lines of the memory, respectively. The control input  $MW$  controls write operations to the memory.

To write a new value into the memory, the address of the desired word is applied to the bus A. In turn, the control input  $MW$  is activated by the CU to enable the write operation. Then, the new value is applied to the bus B. Finally, the value is written to the desired location.

Among those inputs to the datapath and memory, AA, BA, DA, AS, BS, FS,

The elementary operations on the data stored in the registers are typically called microoperations. Microoperations on the datapath include loading a constant value to a register, loading the content of one register to another, adding the contents of two registers and storing the result to another, storing the contents of one register in the memory, etc. In our case, each microoperation completes in a single clock cycle.

*Register transfer language (RTL)* is a convenient notation for representing microoperations. There are total 8 registers in the register file of our datapath. We name each register in the register file as  $R_x$ , where  $x$  is the address of the register in the register file ranging from 0 to 7. That is, we have registers R0, R1, R2, R3, R4, R5, R6, and R7.

RTL statements for data transfer microoperations between registers are in the following form:

where U and V are registers (possibly the same), but V may be a constant. It denotes that the content of a register or a constant V is transferred to a register U. The content of U is overwritten on the positive edge of the next clock cycle, but the content of V remains unchanged. The following is some examples of RTL statements that perform register transfer microoperations and the corresponding control word produced by the CU (a hexadecimal number is prefixed with 0x):

- | AA |   | BA |   | DA |   | AS |   | BS |   | FS |   |   |   | DS |   | RW |   | MW |   | Constant <sub>in</sub> |  | Address <sub>in</sub> |  |
|----|---|----|---|----|---|----|---|----|---|----|---|---|---|----|---|----|---|----|---|------------------------|--|-----------------------|--|
| 0  | 0 | 1  | X | X  | X | 0  | 1 | 0  | 0 | X  | 0 | 0 | 0 | 0  | 0 | 0  | 0 | 1  | 0 | 0xFFFFFFFF             |  | 0xFFFFFFFF            |  |

- | AA | BA | DA | AS | BS | FS | DS | RW | MW | Constant <sub>in</sub> | Address <sub>in</sub> |   |            |            |
|----|----|----|----|----|----|----|----|----|------------------------|-----------------------|---|------------|------------|
| X  | X  | X  | X  | 0  | 1  | 1  | X  | X  | 0                      | 1                     | 0 | 0x00000005 | 0XXXXXXXXX |

- $R5 \leftarrow 0$  ( $R0 \oplus R0 = 0$ )

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>
0	0	0	0	0	0	1	0	0	0XXXXXXXX	0XXXXXXXX

### Arithmetic microoperations

An arithmetic microoperation performs an arithmetic operation provided by the arithmetic unit in the ALU. Its RTL statement has the following form:

$$U \leftarrow V \text{ op}_a W$$

where U and V are registers, and W may be either of a register and a constant. Two or all three of U, V, and W are possibly the same. The binary operation  $\text{op}_a$  is one of + and −. The meaning is that a binary arithmetic operation  $\text{op}_a$  is performed on the contents of V and W (W can be a constant), and the result is transferred to U. The content of U is overwritten, but the contents of V and W remain unchanged. The following is some examples of RTL statements that perform arithmetic microoperations:

- $R0 \leftarrow R1 + R2$  (addition)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>
0	0	1	0	0	0	0	1	0	0XXXXXXXX	0XXXXXXXX

- $R3 \leftarrow R7 - 4$  (subtraction,  $R7 + 4' + 1$ , 4' is the one's complement of 4)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>
1	1	1	X	X	0	1	0	1	0x0000004	0XXXXXXXX

- $R3 \leftarrow R1 + 1$  (increment R1)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>
0	0	1	X	X	0	1	0	1	0XXXXXXXX	0XXXXXXXX

- $R5 \leftarrow R1 - 1$  (decrement R1)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>
0	0	1	X	X	1	0	1	0	0XXXXXXXX	0XXXXXXXX

### Logic microoperations

A logic microoperation performs a logic operation provided by the logic unit in the ALU. An RTL statement that performs a binary logic operation has the following form:

$$U \leftarrow V \text{ op}_l W$$



where U and V are registers, and W may be either of a register and a constant. Two or all three of U, V, and W are possibly the same. The binary operation  $\text{op}_l$  is one of  $\wedge$ ,  $\vee$ , and  $\oplus$ . The meaning of the RTL statement is that a bitwise binary logic operation  $\text{op}_l$  is performed on the contents of V and W (W can be a constant), and the result is transferred to U. The content of U is overwritten, but the contents of V and W remain unchanged.

A unary logic microoperations is bitwise negation ( $'$ ). An RTL statement that performs the bitwise negation operation has the following form:

$$U \leftarrow V'$$

where U is a register, and V may be either of a register and a constant. U and V are possibly the same. The bitwise negation operation is performed on the content of V or a constant V, and the result is transferred to U. The content of U is overwritten, but the content of V remains unchanged. The following is some examples of RTL statements that perform logic microoperations:

- $R0 \leftarrow R1'$  (bitwise NOT)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>										
0	0	1	X	X	X	0	0	0	0	X	0	1	1	X	0	1	0	0	XXXXXXXXXX	XXXXXXXXXX

- $R3 \leftarrow R1 \wedge R2$  (bitwise AND)

AA		BA		DA		AS		BS		FS			DS		RW		MW		Constant <sub>in</sub>	Address <sub>in</sub>
0	0	1	0	1	0	1	1	0	0	0	1	0	1	X	0	1	0	0	0XXXXXXXXX	0XXXXXXXXX

- $R1 \leftarrow R1 \vee R2$  (bitwise OR)

AA		BA		DA		AS		BS		FS				DS		RW		MW		Constant <sub>in</sub>		Address <sub>in</sub>	
0	0	1	0	1	0	0	0	1	0	0	0	1	0	0	X	0	1	0		0xXXXXXXXX		0xXXXXXXXX	

- $R3 \leftarrow R1 \oplus R2$  (bitwise XOR)

AA		BA		DA		AS		BS		FS				DS		RW		MW		Constant <sub>in</sub>		Address <sub>in</sub>	
0	0	1	0	1	0	1	1	0	0	0	1	1	0	X	0	1	0			0xXXXXXXXX		0xXXXXXXXX	

- $R3 \leftarrow R1 \vee 0x0F0F0F0F$

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																	
0	0	1	X	X	X	0	1	1	0	1	0	1	0	0	X	0	1	0	0	0	X	0	0	F	F	F	F	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

## Shift microoperations

A shift microoperation performs a 1-bit shift left or right operation provided by the shifter in the ALU. An RTL statement that performs a shift microoperation has the following form: It has the following form:

$$U \leftarrow \text{op}_s V$$

where U is a register, and V may be either of a register and a constant. The unary operation  $\text{op}_s$  is one of *lsl* (logical shift left), *lsr* (logical shift right), and *asr* (arithmetic shift right). The meaning of the statement is that a 1-bit shift operation  $\text{op}_s$  is performed on the content of a register V or a constant V. The result is transferred to U. The content of U is overwritten, but the content of V remains unchanged. The following is some examples of RTL statements that perform shift microoperations:

- $R0 \leftarrow \text{lsl } R1$

AA	BA	DA	AS	BS	FS		DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>		
X	X	X	0	0	1	0	0	0	X	1	0	0XXXXXXX	0XXXXXXX

- $R2 \leftarrow \text{lsr } R5$

AA			BA			DA			AS	BS	FS				DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>	
X	X	X	1	0	1	0	1	0	0	X	1	0	1	0	X	0	1	0	0XXXXXXXX	0XXXXXXXX

- $R3 \leftarrow \text{asr } R7$

AA			BA		DA		AS	BS	FS				DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>			
X	X	X	1	1	1	0	1	1	0	X	1	0	1	1	X	0	1	0	0XXXXXXXX	0XXXXXXXX

- $R4 \leftarrow \text{asr } 0x80FF0001$

AA	BA	DA	AS	BS	FS			DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>							
X	X	X	X	X	1	0	0	X	1	1	0	1	1	X	0	1	0	0x80FF0001	0XXXXXXXXX

## Memory transfer microoperations

A memory transfer microoperation performs a data transfer operation between the memory and a register. There are two memory transfer microoperations: read and write. A read microoperation transfers a data word within the memory to a register. An RTL statement that performs a memory read operation has the following form:

$$U \leftarrow M[V]$$

where U and V are possibly the same registers. The address of the desired word is given by the content of V. The content of U is overwritten, but the word in the memory remains unchanged.

A write microoperation transfers a data word stored a register to a memory word. An RTL statement that performs a memory write operation has the following form:

$$M[V] \leftarrow U$$

where U and V are possibly the same registers, but U may be a constant. The address of the desired word is the content of V. A write operation makes the content of a register U or a constant U to be transferred to the memory word specified by V. The content of the specified word in the memory is overwritten, but U remains unchanged. The following is some examples of RTL statements that perform memory transfer microoperations:

- $R0 \leftarrow M[R1]$

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>									
0	0	1	X	X	X	0	0	0	X	X	X	X	X	1	1	0	0	XXXXXXXXXX	XXXXXXXXXX

- $M[R2] \leftarrow R4$

AA	BA	DA	AS	BS	FS			DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>		
0	1	0	1	0	0	X	X	X	X	X	0	1	0XXXXXXXXX	0XXXXXXXXX

- $M[R6] \leftarrow 24$

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant <sub>in</sub>	Address <sub>in</sub>		
1	1	0	X	X	X	X	X	X	0	1	0x00000018	0XXXXXXXXX

## 4.4 Programming the Datapath

The datapath can be used for performing various data manipulations including integer computations. For example, we can perform the summation of 10 numbers from 1 to 10:

$$sum = \sum_{i=1}^{10} i$$

The above computation for the datapath can be described with a sequence of RTL statements as shown in Figure 4.6 (a). When we provide the datapath with the sequence of control words that are produced by the RTL statement sequence, the computation completes 22 clock cycles later, and we will have the result in register R0.

Suppose that the 10 numbers are arbitrary integers and they are stored in the memory at consecutive addresses ranging 100 to 109. The sequence of RTL statement shown in Figure 4.6 (b) performs this computation. In this case, it takes 32 clock cycles to obtain the result in register R0.

[illegible]

```

1 sum = 0;
2 i = -1;
3 i = i + 1;
4 sum = sum + a[i];
5 i = i + 1;
6 sum = sum + a[i];
7 i = i + 1;
8 sum = sum + a[i];
9 i = i + 1;
10 sum = sum + a[i];
11 i = i + 1;
12 sum = sum + a[i];
13 i = i + 1;
14 sum = sum + a[i];
15 i = i + 1;
16 sum = sum + a[i];
17 i = i + 1;
18 sum = sum + a[i];
19 i = i + 1;
20 sum = sum + a[i];
21 i = i + 1;
22 sum = sum + a[i];

```

(a)

```

1 sum = 0;
2 i = -1;
3 L: i = i + 1;
4 sum = sum + a[i];
5 if (i < 10) goto L;

```

(b)

Figure 4.7: The C code that computes the sum of 10 integers: (a) the code that does not use a branching mechanism; (b) the code that does use branching mechanisms.

What if we would like to compute the sum of 10,000 arbitrary integers stored in the memory? Then, we need to have a sequence of 30,002 RTL statements. If you carefully take a look at the RTL code in Figure 4.6 (b), the following group of three consecutive RTL statements is repeated 10 times:

$$\begin{aligned}
 R2 &\leftarrow R2 + 1 \\
 R1 &\leftarrow M[R2] \\
 R0 &\leftarrow R0 + R1
 \end{aligned}$$

It would be good if we could have some mechanism to avoid such repetitions. This problem can be solved by using a branching mechanism available in high-level languages, such as C and Java. A branching mechanism alters *control flow*. Control flow refers to the order in which the individual statements are executed.

The computation of adding 10 integers can be represented with the C code as shown in Figure 4.7 (a). A *C statement* is a C expression delimited by a semicolon (;). This code corresponds to the RTL code in Figure 4.6 (b). An *assignment* operation (denoted by =) in C assigns the value of the right-hand operand to the storage location named by the left-hand operand. A *variable* is

a named storage location that contains some value. The variable name typically references the stored value. However, if it is the left-operand of an assignment operation, it refers to the storage location. In the C code, `sum` and `i` are variables whose storage locations are the registers R0 and R2, respectively.

An *array* in C is a collection of elements that have the same *type* (e.g., integers) under the same name. Each element of an array with  $n$  elements can be treated as a variable and is referenced by the array name and an index number ranging from 0 to  $n - 1$ . In the C code, `a` is an array containing 10 integers. The array reference `a[i]` refers to the value of the  $i^{th}$  element in the array `a`. A consecutive group of  $n$  words in the memory is allocated to an array with  $n$  elements.

C statement	RTL statements
<code>sum = 0;</code>	<code>R0 ← 0</code>
<code>i = -1;</code>	<code>R2 ← 99</code>
<code>i = i + 1;</code>	<code>R2 ← R2 + 1</code>
<code>sum = sum + a[i];</code>	<code>R1 ← M[R2]</code> <code>R0 ← R0 + R1</code>

Figure 4.8: The correspondence between the C statements in Figure 4.7 (a) and the RTL statements in Figure 4.6 (b).

Assuming that the storage locations of the variables `sum` and `i` are R0 and R2, respectively, and 100 is the address of the first element `a[0]` in the memory, the table in Figure 4.8 shows the correspondence between the C statements in Figure 4.7 (a) and the RTL statements in Figure 4.6 (b).

An `if` statement in C is a conditional branching mechanism, and it has an expression in parentheses and another statement in the following way:

`if ( E ) S`

where `E` is an expression and `S` is a statement. If the expression `E` is evaluated to a non-zero value, the statement `S` gets executed. It controls *conditional branching*. Depending on the condition `E`, it alters control flow and either the statement `S` or the statement immediately after the `if` is executed. Contrast to the `if` statement, a `goto` statement is a branching mechanism that alters control flow unconditionally. It has the following form:

`goto L;`

where `L` is a label in C. A *label* is a name that identifies a location in source code. The `goto` statement always alters control flow, and control is transferred to a labeled statement (the statement in line 3 of Figure 4.7 (b)) whose label matches the label that appears in the `goto` statement.

Using branching mechanisms available in C, the summation process of 10 arbitrary integers is succinctly described in Figure 4.7 (b). The value of each

element of `a` is continuously added to the variable `sum` continuously until the value of `i` reaches 10.

## 4.5 Instruction Set

A machine instruction is a group of bits that specifies not only the operation, but also the registers or memory words in which the operands are found and the result is stored. The *operation code* (*opcode*) of an instruction is a group of bits in the instruction that specifies an operation. N-bit opcode can represent  $2^n$  different operations. Machine instructions for a computer have either all the same size or different sizes. The way how bits are organized in a machine instruction varies with the type of the instruction and the machine. The *Instruction set* of a computer is the complete collection of instructions for the computer. The *instruction set architecture* (*ISA*) of a computer is a thorough description of its instruction set. The term *microarchitecture* refers to the design techniques used to implement the instruction set.

The instruction set provided by a CPU must be rich enough to implement all functions that are known to be computable. It usually includes the following types of machine instructions:

- Data transfer instructions
  - Load and store instructions that move data to and from memory and CPU registers (load and store instructions).
  - Input and output instructions that moves data to and from CPU registers and I/O devices.
- Arithmetic, logic, and shift instructions.
  - Addition, subtraction, multiply, and division instructions.
  - Bitwise AND, OR, and NOT instructions.
  - Logical and arithmetic shift instructions.
  - Comparison instructions that compare two values.
- Control flow instructions
  - Unconditional branch instructions that jump to another location in the program to execute instructions there.
  - Conditional branch instructions that jump to another location in the program when a certain condition holds.

## 4.6 The Control Unit

The key idea of the von Neumann architecture is the stored program concept. Not only are all data values used in the program stored in memory, but also are

machine instructions in the program. These machine instructions are placed in adjacent locations and fetched by the CU one by one.

A *fetch-decode-execute cycle* is the basic operation cycle of the CPU. The CU fetches an instruction from memory, determines what operations the instruction requires (decodes it), and executes it by activating the necessary sequence of microoperations (i.e., control words) to provide timing and control signals to the datapath and memory. The fetched instruction is decoded by the *instruction decoder* in the CU. The decoder converts the instruction to control signals to the datapath and to the CU itself to perform the operation specified by the instruction. This cycle is repeated until the computer is powered down.

To execute instructions in sequence, it is necessary to provide the address of the instruction to be executed. The CU contains a register called a *program counter (PC)* to specify the address of the next instruction to be executed. The PC is either automatically incremented or loaded with a new address to change the sequence of instructions after an instruction is fetched. *Branch instructions* modify the PC to skip over some sequence of instructions or to go back to repeat the previous instruction sequence. A branch instruction typically contains an offset. This offset is added to the current PC to go to the branch target address. There are two types of branch instructions: conditional branches and unconditional branches. A *conditional branch* instruction modifies the PC when a certain condition is true. The CU evaluates the condition by checking the status signals from the datapath. If the condition is true, the branch is taken. An *unconditional branch* instruction always modifies the PC. It always jumps to the branch target address.

Figure 4.9 shows a simple CPU to which a memory unit is attached. The CU in the CPU controls the 32-bit datapath and memory introduced in this chapter. In addition to a PC, the CU has two other registers called an *instruction register (IR)* and a *processor status register (PSR)*. The IR contains the current instruction fetched from memory. The PSR is used by the CU to keep track of the CPU state. The status signals N, Z, C, and V from the ALU are connected to PSR[31], PSR[30], PSR[29], and PSR[28] as inputs, respectively. These four bits in the PSR are called status flags and named in the same way as the status signals from the ALU. The status flags in the PSR are set by a comparison instruction. The status signals from the ALU indicate the status of the result of the last ALU operation performed by the CPU:

- N (negative): the result of the last ALU operation is negative (MSB = 1)
- Z (zero): the result of the last ALU operation is zero
- C (carry): the result of the last ALU operation has a carry-out
- V (oVerflow): the result of the last ALU operation overflows

As shown in Figure 4.10 (a), assume that a program is stored in the memory at address 0x8000, and the data accessed by the program are stored from address 0xA000. Initially, the PC is loaded with the address 0x8000 of the first



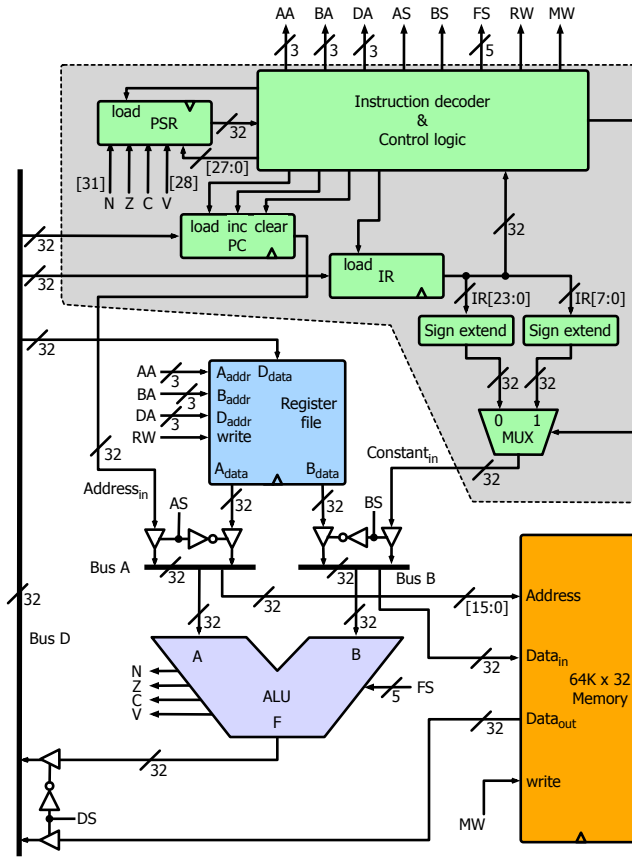


Figure 4.9: A simple CPU with a memory unit.

instruction of the program when power is applied to the system. Then the CPU repeats the fetch-decode-execute cycle.

In the fetch phase, the instruction whose address is specified by the PC is loaded into the IR. This process is the same for all instructions and described by the following RTL statement:

$$IR \leftarrow M[PC]$$

The instruction stored in the IR is decoded by the CU in the decode phase. For example, assume that the current instruction (i.e., the instruction that has been loaded into the IR) is an addition instruction that adds the contents of two registers R1 and R2 and stores the result to the register R3. The instruction has a 32-bit fixed length and four fields for the registers and opcode. Its instruction format is given in Figure 4.10 (b). Rd is a 4-bit field for the destination register R3. Rm and Rn are the two 4-bit fields for the two operand registers R1 and R2, respectively. Since the CPU has a total of 10 registers including the PC and

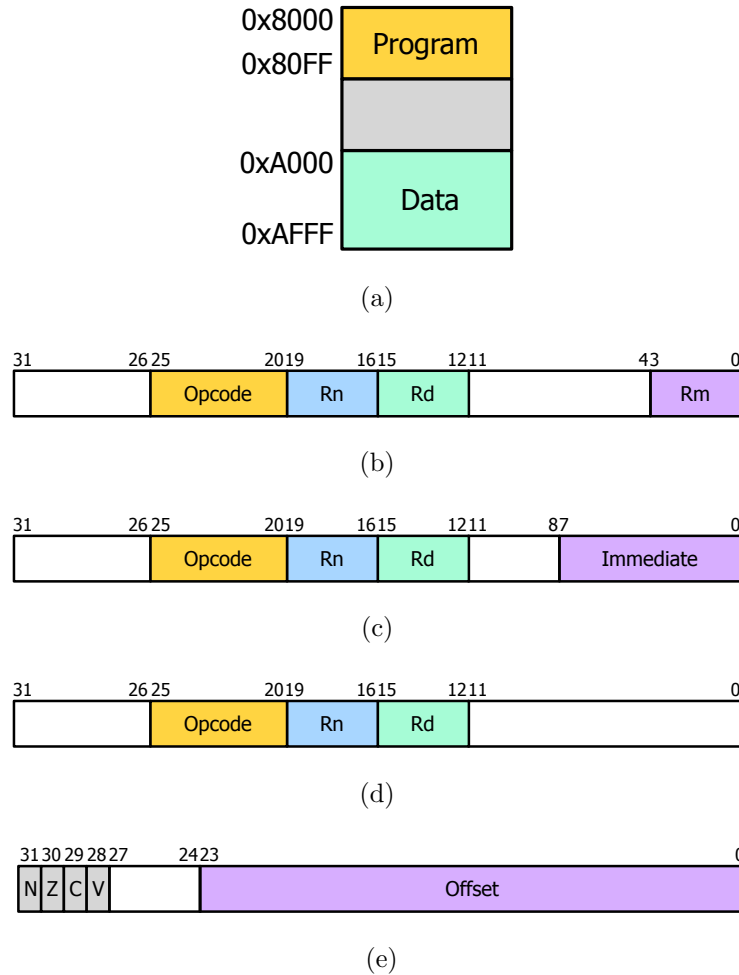


Figure 4.10: Memory map and instruction formats: (a) a memory map for the CPU in Figure 4.9; (b) the instruction format of an addition instruction; (c) the instruction format that contains an immediate constant; (d) the instruction format of a load or store instruction; (e) the instruction format of a branch instruction.

IR, at least 4 bits are required to specify a register. The instruction decoder in the CU reads the content of the IR, and the opcode and operands are being decoded. The CU generates appropriate control words to perform the addition operation in the execute phase:

$$R3 \leftarrow R1 + R2$$

Then, the CU increments the PC to fetch the next instruction:

$$PC \leftarrow PC + 1$$

The CU activates the input *inc* to the PC to perform this increment operation. Since the destination register of the instruction is not the PC, incrementing the PC can be performed in parallel with the addition operation:

$$R3 \leftarrow R1 + R2; PC \leftarrow PC + 1$$

In our RTL notation, RTL statements that are placed in the same line and separated with semicolons (;) are performed in parallel. In summary, the following microoperations are activated by the CU to execute the addition instruction:

$$\begin{aligned} IR &\leftarrow M[PC] \\ R3 &\leftarrow R1 + R2; PC \leftarrow PC + 1 \end{aligned}$$

It takes a total of two clock cycles to perform all the microoperations: one cycle for fetch and another one cycle for decode/execute and incrementing the PC.

Now, assume that the fetched instruction is an addition instruction that adds the content of register R1 and a constant 34, and stores the result to destination register R3. This type of instruction is called an *immediate instruction* because the instruction code contains the actual operand 34. The constant 34 is called an *immediate constant* or an *immediate*. The instruction format is given in Figure 4.10 (c). The immediate field is interpreted as an 8-bit signed binary number in the two's complement representation.

In the decode and execute phases, the immediate field IR[7:0] is sign-extended and connected to the 32-bit input  $Constant_{in}$  of the datapath to execute the immediate instruction. The microoperations activated by the CU are as follows:

$$R3 \leftarrow R1 + Constant_{in}; PC \leftarrow PC + 1$$

A load instruction makes a data word located in the memory to be transferred to a register. A store instruction transfers the content of a register to a memory location. The instruction format of load and store instructions is given in Figure 4.10 (d). The address of the memory location is contained Rn. Rd is the destination register when the instruction is a load instruction. It is the source register when the instruction is a store instruction. When the instruction is a load instruction, the CU activates the following microoperations:

$$Rd \leftarrow M[Rn]; PC \leftarrow PC + 1$$

If the instruction is a store instruction, the following microoperations are activated by the CU in the execute phase:

$$M[Rn] \leftarrow Rd; PC \leftarrow PC + 1$$

As mentioned before, a branch instruction alters the content of the PC. The instruction format of branch instructions are given in Figure 4.10 (e). An unconditional branch instruction alters the PC when non-sequential execution is desired. The specified branch target address is computed by adding the value stored in the *offset* field to the PC. The offset field contains a 24-bit signed

binary number in the two's complement representation. After sign-extending the offset value  $IR[23:0]$ , the CU put the result to the input  $Constant_{in}$  of the datapath. The RTL statement for the unconditional branch instructions is:

$$PC \leftarrow PC + Constant_{in}$$

In the next clock cycle, the instruction located at the branch target address will be fetched. If the destination register of an instruction is the PC, the instruction is treated as an unconditional branch instruction.

When the fetched instruction is a conditional branch instruction, the N, Z, C, and V flags in Figure 4.10 (e) are used to make the decision to take the branch or not. The N, Z, C, and V flags in the instruction ( $IR[31:28]$ ) are compared with the four status flags in the PSR by the CU. If they are the same, the branch is taken, and the following microoperation is activated:

$$PC \leftarrow PC + Constant_{in}$$

Otherwise, the branch is not taken and the next instruction to be fetched is the instruction that follows the current instruction in the memory:

$$PC \leftarrow PC + 1$$

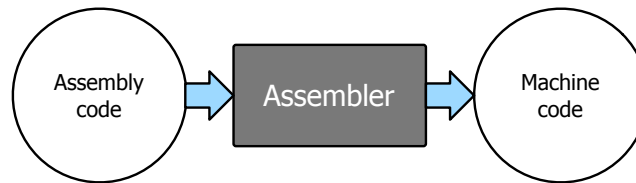


Figure 4.11: An assembler translates the assembly program into the machine code.

## 4.7 Assembly Language

Humans almost never write programs directly in machine code because it is very difficult to understand and write a program in patterns of 0 and 1. It may also be very much error prone because the opcode for every instruction is looked up or remembered to write a program. Instead, we use an *assembly language*. It is a low-level language and relatively easy to write a program compared to the machine language.

Assembly language uses *mnemonics* to symbolically represent the opcode of machine instructions. In addition, it uses symbolic names for locations in the program (labels), variables, and constants. An assembly language instruction usually consists of a mnemonic followed by a list of operands. Since a machine instruction has an equivalent assembly instruction, translation from an assembly

MOV R2, #-25	Move -25 to the destination register R2. An 8-bit integer constant is prefixed with a character #.
ADD R3, R1, R2	Add the value of register R2 to the value of register R1, and stores the result in the destination register R3.
LDR R1, [R3]	Make data located at the address contained in Rn to be loaded into the destination register Rd.
STR R0, [R2]	Make data from the register R0 to be stored to the memory location with the address contained in R2.
CMP R1, R2	Compare the value of register R2 with the value of register R1, and sets up the status flags N, Z, C, and V in the PSR. If the contents are equal, Z is set to 1, otherwise it is set to 0.
B L	Causes a jump to the target address labeled with "L:".
BNE L	Causes a jump to the target address if the Z flag in the PSR is not zero. NE stands for Not Equal.

Figure 4.12: Some examples of assembly language instructions for our computer.

instruction to the corresponding machine instruction is usually straightforward. However, there are some meta-instructions, such as *assembler directives* and *pseudo-instructions*, which may not be translated into a single machine instruction in a straightforward manner. An assembler directive is a command to the assembler that tells the assembler something to do in the assembly process. A pseudo-instruction does not actually exist in the machine instruction set. It is just an easy way of representing a group of machine instructions (possibly a single machine instruction), which makes the code easy to understand. An assembly program may also contains comments that facilitate understanding of the program.

A program written in assembly language is translated into the target computer's machine code by a utility program called an *assembler*. The assembler generates an object file by translating assembly instructions into machine instructions and by resolving symbolic names for memory locations and constants. A label is a symbolic name and specifies a location (address) in the program. When a branch instruction whose address  $a$  uses a label  $L$  as its target, the target address is computed by adding an offset  $L - a$  to the address  $a$  of the branch instruction. The offset is encoded into the offset field of the branch instruction by the assembler.

Figure 4.12 shows some examples of assembly language instructions for our simple computer. The program adding 10 arbitrary numbers stored in the memory can be written in the assembly language. The assembly code is shown in

```

MOV    R0, #0
MOV    R2, #99
L :    ADD    R2, R2, #1
LDR    R1, [R2]
ADD    R0, R0, R1
CMP    R2, #109
BNE    L

```

Figure 4.13: The assembly code that computes the sum of 10 arbitrary integers: (a) the sequence of assembly language instructions that performs the summation of 10 arbitrary integers stored in the memory at consecutive addresses ranging from 100 to 109; (b) the code that performs the same computation with a comparison instruction and a branch instruction.

Figure 4.13. Figure 4.13 (a) is the straight line code, and Figure 4.13 (b) uses a branch instruction. The register R2 is loaded with the address of a number stored in the memory. R2 is incremented from 99 to 109 each time a new value is loaded to R1. The comparison instruction compares the content of R2 with 109. If they are not equal, the branch instruction makes control flow to be transferred to the instruction labeled with L.

Instruction	$T_0$	$T_1$
MOV R0, #0	$IR \leftarrow M[PC]$	$R0 \leftarrow 0;$ $PC \leftarrow PC + 1$
MOV R2, #99	$IR \leftarrow M[PC]$	$R2 \leftarrow 99 (Constant_{in});$ $PC \leftarrow PC + 1$
ADD R2, R2, #1	$IR \leftarrow M[PC]$	$R2 \leftarrow R2 + 1;$ $PC \leftarrow PC + 1$
LDR R1, [R2]	$IR \leftarrow M[PC]$	$R1 \leftarrow M[R2];$ $PC \leftarrow PC + 1$
ADD R0, R0, R1	$IR \leftarrow M[PC]$	$R0 \leftarrow R0 + R1;$ $PC \leftarrow PC + 1$
CMP R2, #109	$IR \leftarrow M[PC]$	$R0 \leftarrow R2 - 109 (Constant_{in});$ $PSR[31 : 28] \leftarrow NZCV;$ $PC \leftarrow PC + 1$
BNE L	$IR \leftarrow M[PC]$	$PC \leftarrow PC + Constant_{in}$ if $PSR[31] = 0$ $PC \leftarrow PC + 1$ otherwise

Figure 4.14: The microoperations that must be performed for each instruction used in Figure 4.13 (b).

Figure 4.14 lists the microoperations that must be performed for each instruction used in Figure 4.13 (b). The comparison instruction is implemented with a microoperation that performs subtraction. According to the result of the subtraction, the status flags in the PSR are set. Note that each microoperation takes a single cycle in the simple computer under discussion. The total time taken to fetch, decode, and execute an instruction is called the *instruction cycle time*. The microoperation to fetch an instruction is the same for all instructions. The CU generates appropriate control words in each clock cycle of the instruction cycle time depending on the opcode identified in the decode phase and the current status of the CPU.

## 4.8 Input and Output

Without input and output (I/O) devices, the computer based on the von Neumann architecture cannot receive information from outside and transmit the

result in a desired form. I/O devices attached to a computer is also called *peripherals*. Peripherals include keyboards, mice, display units, speakers, printers, hard disk drives, optical disk drives, solid state disk drives, network interface cards, etc. Peripherals that communicate with people typically transfer alphanumeric information to and from the CPU. The standard binary code for the alphanumeric information is ASCII.

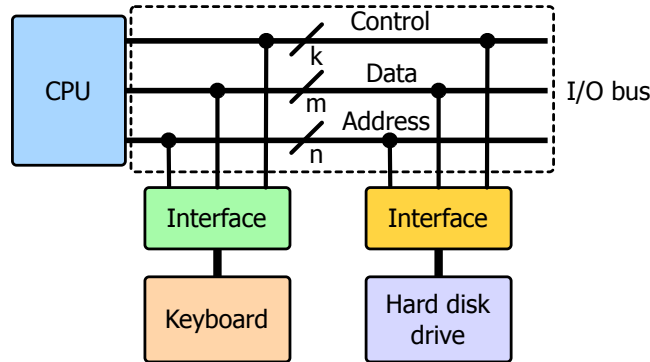


Figure 4.15: The I/O bus.

Peripherals are usually connected to the CPU through an I/O bus as shown in Figure 4.15. The I/O bus consists of data lines, address lines, and control lines. To connect a peripheral to the I/O bus, an *interface* is required to resolve differences between the peripheral and CPU. The interface contains an address decoder, a control unit, and registers for the device.

Each interface has a distinct address. To communicate with a specific peripheral device, the CPU places the address of the device on the address lines, which are continuously monitored by the interface for each device. If the interface for a device detects its own address on the bus, a communication link is established through the I/O bus between the device and the CPU. All other devices are disabled for the bus. The CPU and the selected device communicate with each other through the control and data lines.

## Memory-mapped I/O

*Memory-mapped I/O* makes all I/O devices look exactly the same to the CPU. Each I/O device is allocated to an exclusive portion of the CPU's *address space*. When a CPU has  $n$ -bit addresses, its address space is the set of  $2^n$  possible addresses. The memory and registers of the I/O device have the addresses in the exclusively allocated portion of the CPU's address space. These addresses must not be available for the physical main memory. Each of these addresses is called an *I/O port*. When the CPU accesses a location with an address using a load or store instruction, the location may be a register or a memory location of an I/O device. Since normal load or store instructions are used to communicate with I/O devices, the instruction set of the CPU does not need



to include special I/O instructions. To enable memory-mapped I/O, each I/O device needs to provide a hardware interface similar to that of memory, and is required to define an interaction contract (protocol).

The exclusive portion of the address space allocated to an I/O device continuously reflects the physical state of the device. For example, pressing a key on the keyboard makes a certain value (e.g., ASCII code of the key) to be written in the area allocated to the keyboard. Whenever a bit is changed in the area allocated to a physical screen, the associated pixel is drawn on the screen.



---

## Bibliography

---

- [1] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. Prentice Hall, second edition, 1988.
- [2] Merriam-Webster, Inc. Merriam-Webster Dictionary. <http://www.merriam-webster.com>, 2012.
- [3] von Neumann, John. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15:27–75, October 1993.



---

# Index

---

- ., 16
- abstraction, 12
- algorithm, 13
- ALU, 7
- ANSI, 8
- application, 10
- application software, 10
- arithmetic logic unit, 7
- ASCII, 8
- assembler, 10
- assembly language, 10
- auxiliary stroage, 7
  
- base, 16
- base-10 number system, 16
- binary file, 8
- bit, 6
  - pattern, 6
- byte, 6
  
- C, 11
- C89, 11
- C90, 11
- C99, 11
- compiler, 12
- computer, 5
  - data, 5
  - file, 8
  - input, 5
  - output, 5
  - program, 6
- control unit, 7
- CU, 7
  
- data, 5
- decimal number system, 15
- decimal point, 16
- digit, 15
  
- ENIAC, 8
- executable, 12
- executable object file, 12
- external memory, 7
  
- file, 8
  - binary, 8
  - executable object, 12
  - text, 8
- filename extension, 11
  
- hardware, 6
  
- input, 5
  - device, 7
- instruction, 7
- instruction set architecture, 13
- ISA, 13
- ISO, 11
  
- language, *see* programming language
- least significant digit, 15
- LSD, 15
  
- machine code, 7
- machine instruction, 7
- main memory, 7
- memory, 7
  - external, 7

- main, 7
- non-volatile, 7
- RAM, 7
- random access, 7
- volatile, 7
- microarchitecture, 13
- mixed number, 15
- most significant digit, 15
- MSD, 15
- non-volatile, 7
- number system, 15
  - ., 16
  - base, 16
  - base-10, 16
  - decimal, 15
  - decimal point, 16
  - digit, 15
  - least significant digit, 15
  - LSD, 15
  - most significant digit, 15
  - MSD, 15
  - numeral, 15
  - positional, 15
  - radix, 16
  - radix point, 15
- numeral, 15
- operating system, 10
- OS, 10
- output, 5
  - device, 7
- positional number system, 15
- primary storage, 7
- program, 6
- programmer, 6
- programming, 6
- programming language, 10
  - C, 11
  - high level, 10
  - low level, 10
- radix, 16
- radix point, 15
- RAM, 7
- random-access memory, 7
- secondary storage, 7
- semantics, 11
- service program, 10
- software, 6
  - application, 10
  - system, 10
  - utility, 10
- spreadsheet, 10
- storage, 7
  - auxiliary, 7
  - primary, 7
  - secondary, 7
- stored program, 6, 8
- syntax, 11
- system software, 10
- text editor, 11
- text file, 8
- tool, 10
- Unicode, 8
- utility, 10
- utility software, 10
- volatile, 7
- von Neumann architecture, 6