CPT 101

Computer Systems

By Baby Rui Cry Cry

2021.01.18

1. Overview (总览)

Types of Computers

- Mainframe computers (1960s)
- Supercomputers (1970s)
- Workstations (1980s)
- Microcomputers (1980s)
- Personal computers (1980s)
- Microcontrollers (1980s) **(Downsizing)**
  - embedded or dedicated computers: from calculators to automobiles
- Servers (1980s) **(Increased performance)**
  - Network, Web, Cloud
- Chip computer (?)

Computer Generations

- First (1944 to 1958): **Vacuum tube**
  - MARK I, ENIAC and UNIVAC
- Second (1959 to1963): **Transistor**
  - IBM1401, IBM1410 with 1402 card
- Third generation (1964 to 1970): **IC**
  - DEC PDP 1 and IBM 360

Computer Hardware

5 categories

1. Input
2. Processing
3. Output
4. Storage
5. Communications

Computer Software

- System software
  1. communication with hardware
  2. resource management
  3. facilitates communication among application programs
- Applications software
  1. benefits or assists the user

Backward (Downward) Compatibility for new hardware (新硬件向下兼容)

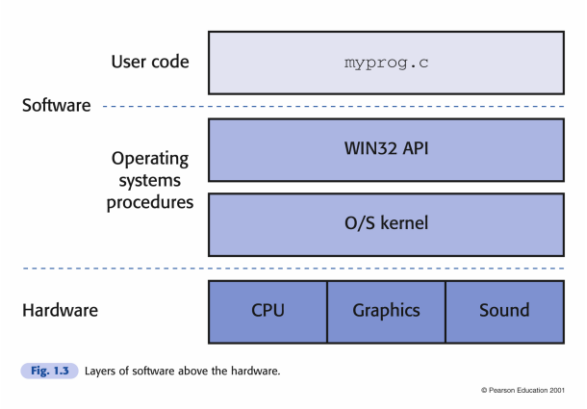
Most software written for computers with old hardware can be run on computers with newer hardware

VHDL (超高速集成电路硬件描述语言)

1. Very high-speed integrated circuits Hardware Description Language.

2. A programming language to be used to specify both the structure and function of hardware circuits.
3. Supports computer simulations as well as providing input to automatic layout packages which arranges the final circuits.

Hierarchy of systems (分层控制系统)



Advantages of the operating system

- Functionalities of hardware systems can be brought out by operating systems and thus offered to the user.
- The user's programs interact with hardware systems through the functionalities provided by operating systems.
    1. Ease of programming.
    2. Protection for the system and for other users.
    3. Fairness and efficiency of using system resources

Hardware evolution: Moore's Law (硬件革新：摩尔定律)

Gordon Moore (January 3, 1929) One of the founders of Intel Corporation.
A circuit designed 24 months ago can now be shrunk to fit into an area of half the size. (It is sometimes quoted as every 18 months.)

Interacting development of hardware & software: Examples

1. WIMPs (Windowing interfaces)
   A by-product of the microprocessor revolution, which allowed all users to have fast bitmapped graphics on their desks.
2. The Internet
   The launch of Netscape browser boosted the use of Internet greatly!

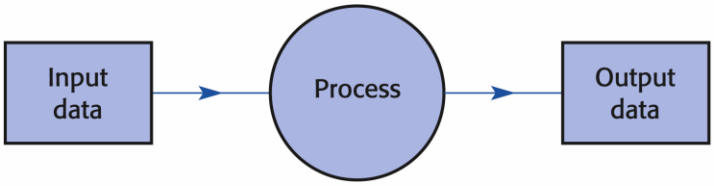
Trends of Computing

1. Scientific computing: Computation
2. Business computing: Data
3. Personal computing: Interaction
4. Pervasive computing: Ubiquity
5. Mobile computing: Mobility

2. I-O Process, von Neumann Model

(输入-过程-输出模型，冯•诺伊曼结构)

Input Process Output Model (输入-过程-输出模型)



1. The Input Process Output Model is the fundamental structure of the current generation of digital computers.
2. The process is to be controlled by a special, custom made program.
3. This was an essential scheme of the von Neumann model.

Components of the Computer System

There are three components required for the implementation of **Input Process Output and von Neumann model(s)**:

1. Hardware.
2. Software.
3. Data that is being manipulated.

Hardware

1. **Central Processing Unit (CPU)**
   An active part which performs calculations and other operations.
2. **The main memory** (primary storage or working storage) or **RAM** (for random access memory)
   They hold data and programs for access by CPU.
   They are volatile.
3. **The secondary storage**
   Long term storage.
   Holds programs and data.
   Hard disk, CDs, DVD, etc.
4. **Input devices**
   Keyboard, mouse, scanner, etc.
5. **Output devices**
   Monitor, speaker, printer, etc.

Software

The hardware of a computer (e.g., CPU) can carry out only very simple operations like adding numbers (very quickly).

To make it perform useful tasks, these simple steps are combined in the form of programs, which are collectively known as software.

Machine instructions (机器指令)

1. The CPU performs the execution of machine instructions.
2. Every CPU has its own instruction set (100-200 instructions, typically).
3. For a particular machine, this set is fixed.
4. Although the instruction sets of different CPUs are similar, there is no standard instruction set.

Machine Instruction Categories

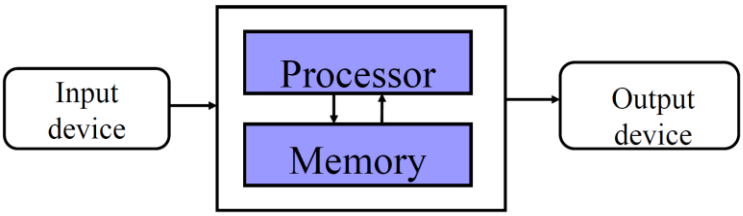
1. **Input output**
   IN OUT (Intel x86 and Pentium, but does not exist in some CPUs)
2. **Data transfer and manipulations**
   MOV ADD MUL AND OR
3. **Transfer of program control**
   JMP, JC
4. **Machine control** (can halt processing, reset the hardware)
   INT, HLT

Machine Instructions and HLL (高级语言)

High **L**evel Programming **L**anguages (HLLs) are more suitable for programming than the languages of machine instructions.
The programs in HLL still have to be translated to the machine codes.
Examples: FORTRAN, COBOL, C, C++, Java, Perl, …

The von Neumann Model

The idea was formulated by von Neumann (late 1940s).
The computer is a general-purpose machine controlled by an executable program.



1. A **program** is a list of instructions used to direct a task.
2. Both program and data are held in computer's **memory** (store) and both represented by **binary codes**.
3. The fact that memory is re-writeable makes a von Neumann machine especially powerful.
4. A **processor** is an active part of the machine that executes the program instructions.
5. Input device is for transmitting information from a user into the computer's memory.
6. Output device enables a user to see results of the program being performed.

Von Neumann bottleneck:

CPU is continuously forced to wait for vital data (and instructions) to be transferred to or from memory.

Potential problems:

1. How could computers distinguish data from instructions since they are both represented by binary codes?
2. A 16-bit instruction code could, in different circumstances, represent a number or two characters.

Solution:

1. Data and instructions are stored in memory.
2. CPU knows where to fetch program instructions.
3. Central Processing Unit (CPU) executes the program instructions.
4. Instructions and data have to be in a special coded form in order to be understood by CPU.

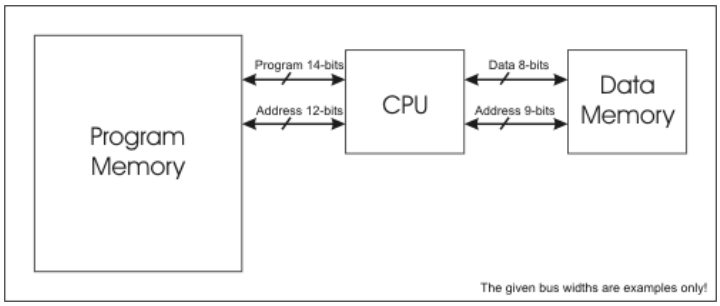
History:

Von Neumann's specification has remained sound for more than 60 years and is implemented in almost all computers today.

Harvard architecture (哈佛架构)

A variation, "Harvard architecture", has gained ground recently.

1. Separates data from programs.
2. Requires different memories and access buses for programs and data.
3. The intention is to increase transfer rates, improving throughput.



Comparison between two model

1. **The Von Neumann Architecture** has won: Most memory (hard drives and RAM) can hold data as well as instructions.
2. **The Harvard Architecture** is still going strong: Most desktop CPUs have an internal "instruction cache" feeding the control unit and a completely separate "data

3. Machine instruction, HLL, compilers

(机器指令，高级语言，编译器)

Semantic gap (语义间隙)

The term expresses the enormous difference between the way human languages expressing ideas and actions and the way computer instructions representing data processing activities.

Translation

Translation is done by special programs such as:

1. **Compilers (编译器)**

   Translating HLL instructions into machine code (sequence of instructions) before the code can be run on the machine.

2. **Assemblers (汇编器)**

   Translating mnemonic form of machine instructions (like MOV, ADD, etc) into their binary codes.

3. **Interpreters (解释器)**

   Translating HLL instructions into machine code on the fly (while the program is running).

Translation compilers and assemblers

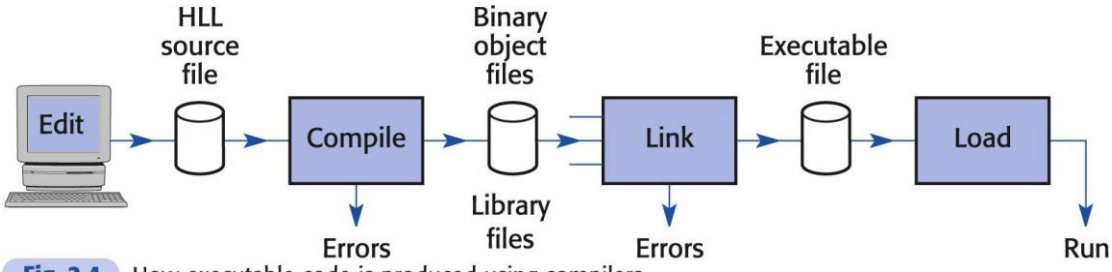


Fig. 2.4 How executable code is produced using compilers.

Linking: resolving external references.

Linking (链接)

Background:

1. Big programs usually are divided into several separate parts or modules.
2. Each module has to be designed, coded and compiled.
3. There are frequent occasions when code in one module needs to reference data or subroutines in another module.
4. A compiler can translate a module into binary codes, but it cannot resolve those references to other modules. Those external references remain symbolic after the compilation, until the linker gets to work.

Function:

1. The linker is to join together all the binary parts.
2. The linker will report errors if it cannot find the module or code referred to by those

external references.

Diagram:

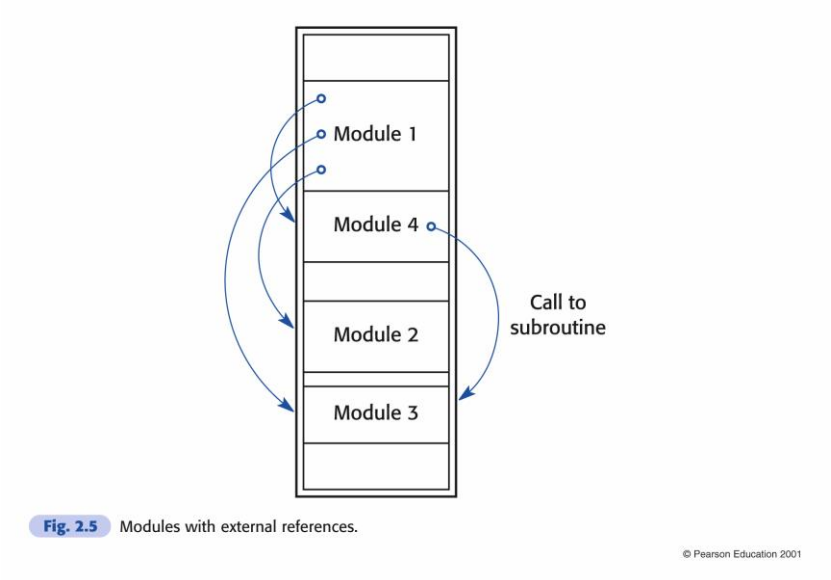


Fig. 2.5 Modules with external references.

Library files (库文件)

1. Translated object code.
2. Provide many functions for programmers, but are only usable if linked into your code.

Example:

In Unix: Directories /lib and /usr/lib/.

In Windows: DLL files.

Interpreters

Alternative way of running HLL programs

Examples:

Such as used with BASIC and Java.

Diagram:

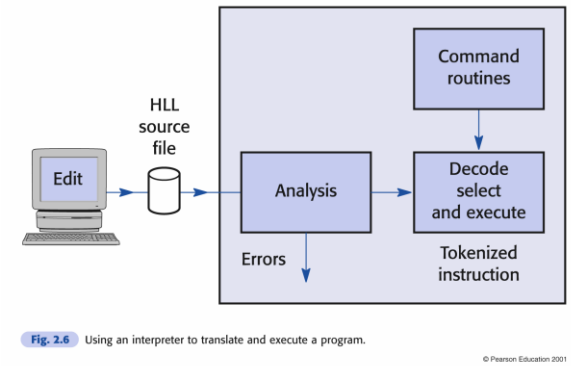


Fig. 2.6 Using an interpreter to translate and execute a program.

Principle:

1. Instructions are converted into an intermediate form, consisting of tokens (In Java, tokens such as: static, boolean, file, string, void, return)

2. Tokens are then passed to the decoder, which selects appropriate routines for execution.

Comparison between Compilers and Interpreters

Compilers:

1. Take a program and translate it as a whole into machine code.
2. The processes of translation and execution are separate.

Interpreters:

1. Take an instruction, one at a time, translate and execute it.
2. The processes of translation and execution are interlaced.

Example:

C program compilation, linking & execution:

C language source code → Compiler (program) → Assembly language → Assembler → Machine code → Linking and loading (program) → Program code execution (program)

Java compilers and interpreters:

Java source code → compiler (program) → Java "byte codes" → Java interpreter (program)

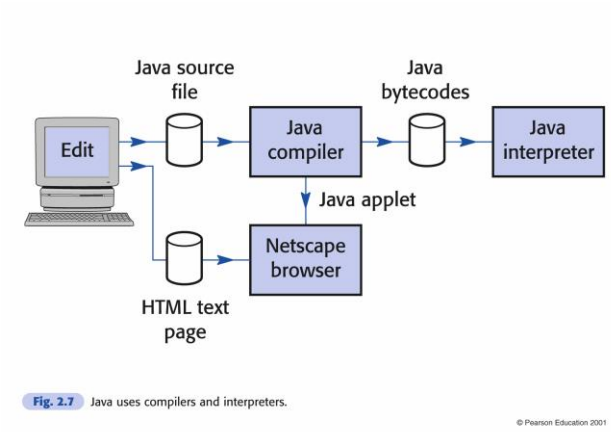


Fig. 2.7 Java uses compilers and interpreters.

Interpreters vs. Compilers

Compilers:

1. Execution of compiled code is much faster than execution of interpreted code.
2. But this disadvantage of interpretation is not a big problem for most applications.

Interpreters:

1. Interpreters are more suitable for rapid prototyping and for other situations when a program is frequently modified.
2. Interpreters are more accurate in terms of error reporting.
3. Interpretation can provide uniform execution environment across several diverse computers. (Portable)

Interpreters as Virtual Machines (程序虚拟机)

Interpreters are somewhat similar to the computer hardware (CPU), as they take one instruction at a time and execute it, because sometimes they are referred to as a virtual machine.

Example: JVM, Java Virtual Machine.

Code sharing and reuse

The problem:

How to reuse existing proven software when developing new systems?

Three solutions:

1. Source level subroutines and macro libraries.
2. Pre translated re locatable binary libraries.
3. Dynamic libraries and dynamic linking.

Diagram:

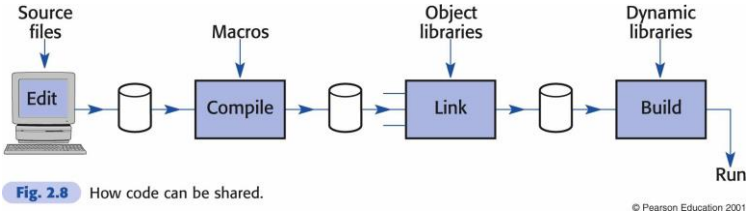


Fig. 2.8 How code can be shared.

Source level subroutines and macro libraries

Intention:

1. Take copies of the library routines.
2. Edit them into your new code.
3. Translate the whole together.

Disadvantages:

1. Who owned the code?
2. Who should maintain it?

Pre-translated relocatable binary libraries (预编译可重置二进制库？)

Intention:

1. Libraries are pre translated into relocatable binary code
2. Can be linked into your new code, but not altered.

Acceptance:

1. Successful, and still essential for all software
2. development undertaken today.

Disadvantage:

1. Each program is to have a private copy of the subroutines, wasting valuable memory space, and swapping time, in a multitasking system.

Dynamic libraries and dynamic linking (动态链接库)

Intention:

1. Load a program which uses "public" routines already loaded into memory.
2. The memory resident libraries are mapped, through the memory management system to control access and avoid multiple code copies.

Acceptance:

1. Successful through Microsoft's ActiveX standard.

4. Data code, hex, data conversion (编码，进制转换)

Data, Information and Knowledge

Data:

Raw facts, figures, measurements, …

1.00001, 1.00000010, 2.0000101, 3.0000102, 5.000, …

Information:

Data organized into useful representation.

1.000, 1.000, 2.000, 3.000, 5.000, …

Knowledge:

Application of reasoned analysis of information.

'Data are in increasing order', 'data can be derived based on Fibonacci sequencing', etc

Measurement of Information

1. Claude Shannon's ideas created two main lines of development: information theory and coding theory.
2. Entropy of an event X is related to p(X).

Bit (比特)

A bit is the most basic unit of information possible: it contains the information necessary to distinguish two alternatives (1 or 0, YES or NOT, etc.).

We think of these alternatives as being numbers 0 and 1

Two alternatives are represented by two state elements (memory cell is charged, or not).

Binary number system (二进制数系)

Definition:

A sequence of bits (binary digits) represents a number in the binary notation.

As in decimal system a binary digit (1 or 0) represents some value depending on where it is written in the number.

Example:

1024	512	256	128	64	32	16	8	4	2	1
1	0	1	1	0	0	1	0	0	1	1

1024 + 256 + 128 +16 +2 +1 = 1427.

Therefore, 10110010011 is a binary representation of 1427 (decimal).

Conversion of binary numbers to decimal (二进制与十进制转换)

1. Multiplying the weighting value by the associated digit (bit) and
2. Adding the results.

Binary vs. decimal notation

1. Decimal notation is more compact than binary.
2. Decimal is more convenient for humans.
3. Binary is more "convenient" for computers due to two state, ON/OFF technology

employed.

Notation

Every number can be used as a base.

1. Decimal notation uses 10 as a base.
2. Binary notation uses 2 as a base.
3. Hexadecimal notation uses 16 as a base.
4. Octal notation uses 8 as a base.

Hexadecimal notation (十六进制表示法)

Advantages:

1. Convenient shorthand for binary numbers.
2. Every hexadecimal digit exactly represents 4 binary bits (binary number) and vice versa.

$0 = 0000_2$	$8 = 1000_2$
$1 = 0001_2$	$9 = 1001_2$
$2 = 0010_2$	$A = 1010_2$
$3 = 0011_2$	$B = 1011_2$
$4 = 0100_2$	$C = 1100_2$
$5 = 0101_2$	$D = 1101_2$
$6 = 0110_2$	$E = 1110_2$
$7 = 0111_2$	$F = 1111_2$

Translation from binary to hexadecimal:

1. First breaking it into 4 digits groups (from right to left) and
2. Translating each group into one hexadecimal digit:
   $10011110_2 = 9E_{16}$

Translation from hexadecimal to binary:

1. Translate every hexadecimal digit into the 4-digit binary pattern.
   $AA3E_{16} = 1010101000111110_2$

5. Char, ASCII, Unicode (字符，ASCII 编码，Unicode 编码)

Alphanumeric characters (字母数字字符)

Data that computers deal with usually comprise:

1. Letters, such as 'A', 'B', etc.
2. Numbers, such as '1', '9', etc.
3. Other characters, such as punctuation marks.

Alphanumeric codes (字母数字编码)

1. **ASCII code** (American Standard Code for Information interchange) (7-bit code) and its extensions (8-bit codes) (well-established).
2. **EBCDIC code** (Extended Binary Coded Decimal Interchange Code) 8-bit code. (IBM mainframe computers)
3. **Unicode**. Recent 16-bit standard. (Up to 2^{16} characters can be encoded)

ASCII code

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Characteristics:

1. It is 7-bit code, so the code of any character can be represented as a two-digit hexadecimal number (row's and column's numbers are the first and second digits, respectively).
2. 3 bits are represented by the first hexadecimal digit and the next 4-bits are represented by the second hexadecimal digit.
3. Only half of possible byte (8 bits) patterns is used.
4. The second half is used in the 8-bit extension(s) to represent the codes of additional symbols, line shapes, foreign letters, etc.
5. The order of the letter codes is compatible with the alphabetical order of the letters.
6. The table is divided into two classes of codes:
   a) Printing characters.
      Printing characters produce output on the screen or on a printer.
   b) Control characters.
      i. To control the position of output on the screen or paper (e.g., 'HT').
      ii. To cause some action to occur (e.g., 'BEL').
      iii. To communicate status between the computer and an I/O device (e.g.,

'Control C' combination).

Limitation:

The limitations of the well-established 8-bit ASCII codes.

1. Too limited for the display requirements of modern Windows based word processors.
2. The requirement of global software market for handling international character sets.

Unicode

Unicode Standard (1991) is a 16-bit international encoding system for information interchange. Code values are available for more than 65,000 characters.

Character set:

1. Standard includes the European alphabetic scripts, Middle Eastern right to left scripts, and scripts of Asia, Africa and America, ideographic characters of China, Japan, etc.
2. Standard includes punctuation marks, mathematical symbols, geometrical shapes, etc.

Representation of numbers

Any representation of numbers capable of supporting arithmetic manipulation has to deal with both integers and real values, positive and negative.

1. Integers are whole numbers, with no fraction part.
2. Real numbers extend beyond the decimal point.

Two's complement:

1. Generally, 4 bytes, i.e., 32 bits.
2. Two's complement as a method of representing and manipulating negative integers.

-128	64	32	16	8	4	2	1

$00000101_2 = 5_{10}$ $11111011_2 = -5_{10}$

IEEE 754 standard:

1. The most widely used standard for floating point computation.
2. Defines formats for representing floating point numbers, special values, and a set of floating-point operations that operate on these values.

Declaration of variables in programs

1. You are telling the compiler to reserve the correct amount of memory space to hold the variable.
2. You are also telling the compiler what encoding/decoding/representation scheme to be used.

6. OS and Net

(操作系统和网络)

Operating systems (操作系统)

Operating systems have undergone more than 40 years of evolution, but surprisingly little has changed at the technical core. Only the introduction of windowing interfaces really distinguishes.

Functions:

Functions of OS to interpret and carry out commands issued by:

1. Users of the computer.
2. Application programs running on the computer.

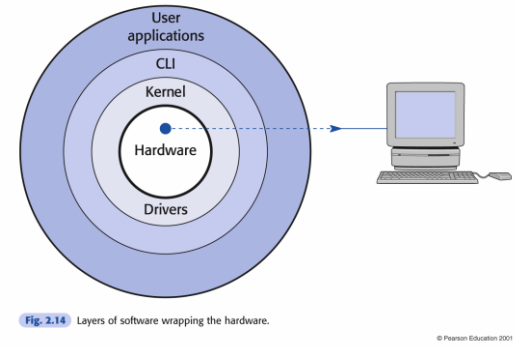
Purposes:

Two fundamental purposes:

1. **Management:**
   a) To control and operate hardware in an efficient way.
2. **Provide functionalities:**
   a) To allow the user efficient access to the facilities of the machine.
   b) To allow the user fair access to the facilities of the machine.
   c) To allow the user protected access to the facilities of the machine.
   d) Interaction with the user.

Onion ring model:

1. To realize the purposes of an operating system, the software system is often arranged in multiple layers.
2. We use a computer through the operating system. (gatekeeper)
3. It is often the operating system that governs the overall efficiency of a computer. (butler)
4. Core of operating system: dealing directly with the hardware.
   a) Kernel: device drivers, memory allocator…
   b) CLI: provide user accessibilities to the system.



Modern operating Systems

Many modern operating systems also:

1. Allow for the simultaneous processing of multiple programs.

DOS:

a) To run a second program, you have to wait the current one finishes.
b) Background spooling provides minimal degree of concurrency.

Windows:

a) Programs can run simultaneously, if they are not too resource consuming. (Same with Unix and Linux.) (Multi-tasking)

2. Support for multiple users. (Multi-user)

Interaction with operating system

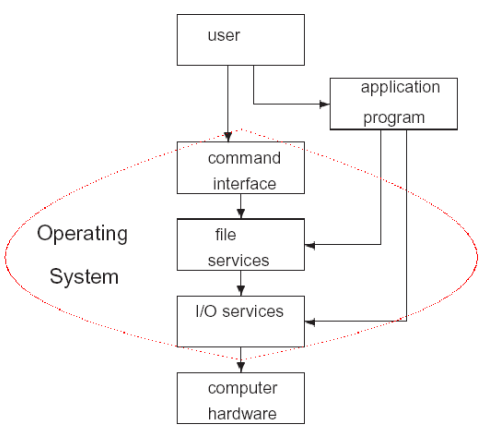
1. CLI (command line interpreter)
   a) DOS: type a command in a command line.
   b) Unix/Linux: shell scripts (sequences of instructions).
   c) Windows/Mac OS X: click with mouse on icons.
2. Function calls from within user programs (API)

Complexity of OS operation

If multiple programs are executing simultaneously, OS must include also:

1. Program to allocate memory and other resources to each program. (**memory manager**)
2. Program to allocate CPU time to each program (**scheduler**: more details later)
3. Program to maintain integrity of each program. (**security kernel**)

Simplified picture of OS services



Computer Networks

Description:

1. Perhaps the most far-reaching changes ever produced to von Neumann's original blueprint.
2. Operating system usually provides access to network facilities. (via networking API, e.g., socket interface)
3. Computer network is an interconnected collection of autonomous computers to facilitate fast information exchange.

Advantages:

1. To increase computing power.

2. To share valuable resources (printers, large disks facilities, programs, data bases, etc).
3. To organize convenient interactions of users working at their own computers, often from different locations.
4. Information at your fingertips…

Connectivity

1. Connecting devices for communication
   a) foundation for information age
2. E-mail
   a) send and receive messages over a local area network or a large network, including the Internet.
3. Telecommuting
   a) Working at home or on the road.
   b) Communicating with the office through phone, fax, and/or computer.
4. On line shopping, E or M commerce

Browsers and the World Wide Web

Two dominant Web browsers:

1. Microsoft Internet Explorer
2. Netscape Navigator

Client server computing

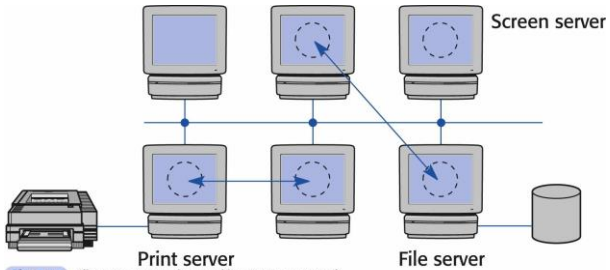


Fig. 2.15 Client–server couplets working across a network.

Client: The originator of a request.

Server: The supplier of the service.

Client server interaction (客户端服务器交互)

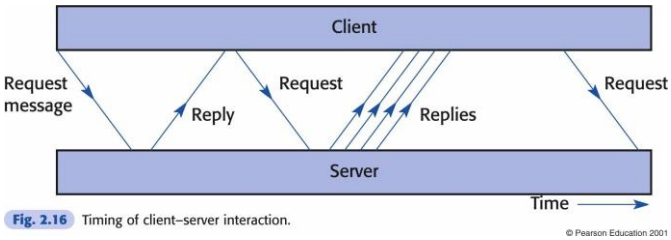


Fig. 2.16 Timing of client–server interaction.

1. Client starts the interaction by sending a request message to the server.
2. Server responds by sending replies back…
3. Look at the status bar of your web browser when opening a web page from a remote site.

7. Instruction cycle (指令周期)

The principal components of a computer

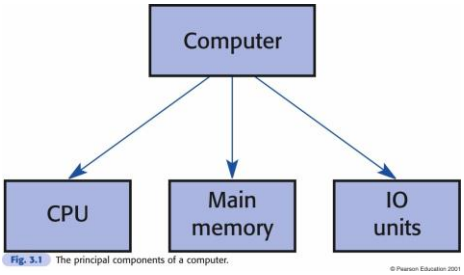


Fig. 3.1 The principal components of a computer.

1. These are the minimum set of components for a working digital computer.
2. A PC motherboard often appears much more complicated.

Motherboard (主板)

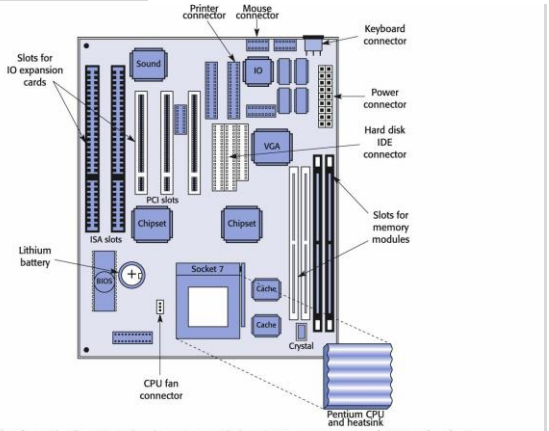


Fig. 3.2 PC-Atmotherboard, showing the locations of the CPU, memory and IO cardsockets.

1. Three principal subsystems:
   a) CPU
   b) main memory
   c) input output units
2. Each of these is often made up of many components.

Processor and Registers (处理器和寄存器)

Processor:

1. **Arithmetic/logic unit (ALU) (算术逻辑单元)**
2. **Control unit (控制器)**: Part of a CPU responsible for performing the machine cycle - fetch, decode, execute, store.

Registers:

1. **Program counter (PC) (程序计数器)**: Contains the address of the next instruction to execute.
2. **Instruction register (IR) (指令寄存器)**: Part of a CPU control unit that stores an instruction.

Coprocessors (协处理器): Assistants to the CPU

1. Coprocessors: microprocessors performing specialized functions that CPU cannot

perform or cannot perform as well and as quickly.

a) Math
b) Graphics

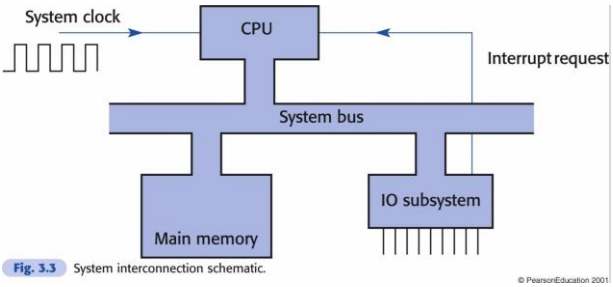
Boards and Chips

1. Circuit boards
2. Use aluminium or copper to conduct electronic messages
3. Chips of silicon
4. Semiconductor

Buses (总线)

1. On the motherboard, all the components are interconnected by buses ("signal highways").
2. A bus is a bundle of conductors, wires, or tracks.
3. Typically, there are **address**, **data** and **control** buses, each including several signal lines.
4. Each hardware unit is connected to all these buses.
   a) A simple way of building up complex systems in which each unit can communicate with each other.
   b) Little disruption when plugging in new units and swapping out failed units.

System interconnections



1. The connected devices can have access to any signal line they require.
2. Bus interconnection is often represented in diagrams as a wide pathway, rather than showing the individual wires.

Bus vs. point-to-point connections

An alternative scheme: **point-to-point interconnection**.

1. The number of pathways needed to link every possible pair of n units: $n(n-1)/2$.
2. Each pathway will still require a full width data highway.
3. The result will be a huge number of wires.

Pros and cons:

1. The number of wires required for a bus is much smaller than that for point-to-point connections.
2. A bus can only transfer one item of data at a time, like a railway line.
   a) Leads to a limit on the performance, termed the **Bus Bottleneck**.
   b) It cannot be solved by simply increasing the speed of a processor.

Registers

1. CPU registers: small block of fast memory.
   a) Temporarily store for data and address variables.
2. Some CPU registers:
   a) **Instruction Pointer (IP)** or Program Counter (PC).
      - Stores the address of the next instruction.
   b) **Accumulator (AX, EAX in Pentium)**.
      - General purpose data register.
   c) **Instruction Register (IR)**.
      - Stores the instruction that is being executed.
3. **Memory address register (MAR)**
   - Temporarily holds address of the memory location during a bus transfer.
4. **MBR**

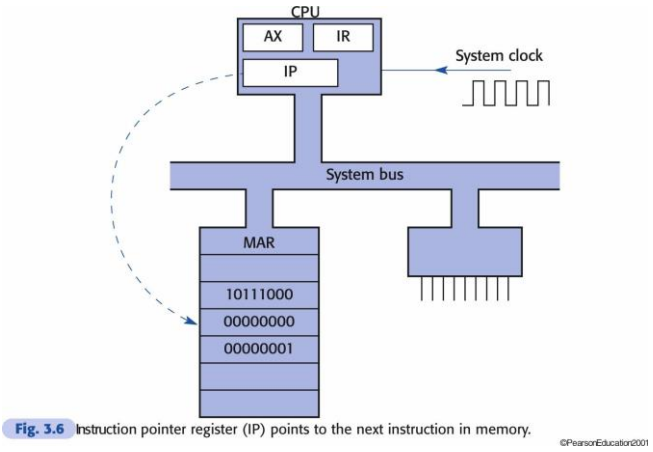


Fig. 3.6 Instruction pointer register (IP) points to the next instruction in memory.

Instruction Set (指令集)

The collection of machine language instructions that a particular processor understands.
instructions for a specific CPU:

1. Designed to be executed by a computer without being translated.
2. Also called machine code.
3. Operations like: ADD, SUB, INC, DEC, etc.

Machine cycle (机器周期)

The basic operation, known as the fetch execute cycle or machine cycle.
The sequence whereby each instruction of the program is executed:

1. Read from the memory.
   - Fetch the instruction from memory. This step brings the instruction into the instruction register, a circuit that holds the instruction so that it can be decoded and executed.
2. Decoded.
   - Read the effective address from memory if the instruction has an indirect address.

3. Executed.
    - Store the result.

The fetch phase of the cycle (读取阶段)

1. The address in IP register is copied onto the address bus and further to MAR register.
2. IP is incremented ready for the next cycle. IP now points to the next location in the program memory.
3. Memory selects location and copies the content onto the data bus.
4. CPU copies the instruction code from the data bus into IR.
5. Decoding of the instruction starts.

Example:

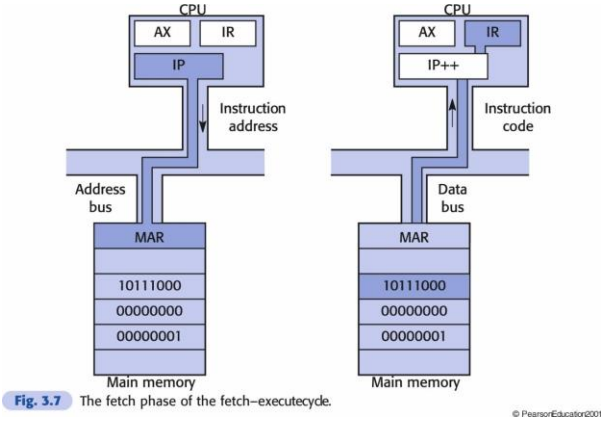


Fig. 3.7 The fetch phase of the fetch–executecycle.

A Pentium instruction: 10111000 00000000 00000001
Assembly code: MOV AX 0x100
Note that the content of a memory cell is different from its address (not shown in the figure).

The execution phase of the cycle (执行阶段)

Execute phase depends on the type of instruction.

Example:

The execution of MOV AX, 256 instruction includes:

1. IP is copied to address bus and latched into memory.
2. IP is incremented.
3. The value selected in memory is copied onto the data bus.
4. CPU copies the value from the data bus into AX.

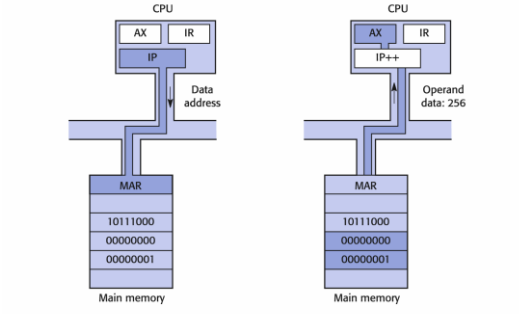


Fig. 3.8 The execution phase of the fetch–execute cycle for MOV AX, 256.

CISC & RISC: (复杂指令集和精简指令集)

1. CISC ("sisk")
   a) complex instruction set
   b) most mainframes and PCs
2. RISC ("risk")
   a) reduced instruction set
   b) cheaper and faster
   c) shift some work to software

In RISC an instruction usually consists of a single word.
But, in CISC an instruction may be several words long, requiring several fetches.

RISC is faster because:

1. The vacated area of the chip can be used to accelerate the performance of more commonly used instructions, rather than compensating for those rarely used instructions.
2. Easier to optimize the design.
3. Simplifies translation from high level languages into the smaller instruction set that the hardware understands, resulting in more efficient programs.

Output Hardware (输出硬件)

1. Hardcopy output
   a) Graphics
   b) Letters
2. Softcopy output
   a) Video
   b) Audio

Screen Clarity (屏幕分辨率)

Standard screen resolutions (in pixels):

1. 640 x 480
2. 800 x 600
3. 1024 x 768
4. 1280 x 1024
5. 1600 x 1200
6. ……

Communications Hardware (通信硬件)

Facilitate networks:

1. Modems
2. Hubs and other components of a network

Ports (端口)

Connecting peripherals to the computers

1. Parallel port (IEEE 1284)

    - printers, some scanners
2. Serial port (RS 232)
    - modems, scanners, mice

USB (Universal Serial Bus) (通用串行总线)

1. Industry standard developed in the mid-1990s that defines the cables, connectors and protocols used for connection, communication and power supply between computers and electronic devices.
2. Standardized the connection of computer peripherals, such as keyboards, pointing devices, digital cameras, printers, portable media players, disk drives and network adapters to PCs.
3. Replaced earlier interfaces, such as serial and parallel ports, as well as separate power chargers for portable devices.

Power supply (电源)

Protected by power surge protector or Uninterrupted power supply unit (UPS).

8. Assembly language and assembler, instruction exec

(汇编语言，汇编器，指令执行)

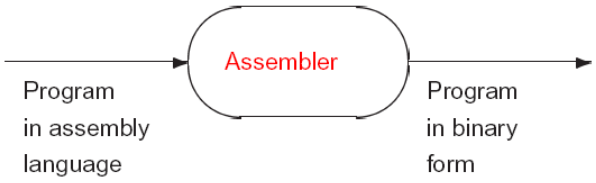
Mnemonic form and assembly language (助记码和汇编语言)

In order to study more closely the way in which programs are performed, we will write some simple programs for **the Pentium processor**. We will write this in **a mnemonic form** of the Pentium machine code. This form is called assembly language.

Assembly language and assembler

The assembler program takes as input a program written in assembly language:

1. Translate instructions into their binary machine code form.
2. Associates labels with memory addresses.
3. Produces a binary machine object code program.



Associates labels with memory addresses: (将标签与内存地址关联)

Example:

```
L1:    JZ  L2
       …
       CALL doD
       JMP L1
L2:    …
```

1. Labels, for example, signify the beginning of a loop, the beginning of a block of memory cells, etc.
2. A program has to be loaded into somewhere in main memory in order to be executed.
3. Programs must be **relocatable**. **(程序必须是可再定址的。)**

Binary machine object code program:

1. Also called object file.
2. Contains binary machine code.
3. May still need a linker to generate an executable or library by linking object files together.

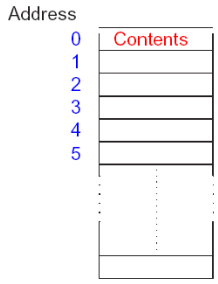
Main memory (RAM) (主存)

A CPU may execute only instructions loaded in the main memory.

RAM: Random Access Memory (随机存取存储器)

The name is to set it apart from sequential access memory (e.g., serial tape storage) with which you cannot access data stored in the last blocks without going through the first blocks of data.

1. The memory can be seen as a set of numbered storage elements, called **words (字)**, each of which can contain some information.
2. Each word is numbered with its **address (地址)**.
3. Any word of memory can be accessed "without touching" the preceding words (Random Access).
4. Access time is the same for all the stored items.



Words, Bytes and Bits (字，字节和比特)

1. A **word** may be visualised as a set of more elementary storage elements (memory cells), called **bits**, arranged in a row.
2. The number of bits in the word may vary between different computers; **32 bits** in **word** is common.
3. A standard unit of **8 bits** called a **byte**.

EAX register, or Accumulator

1. Used as a general-purpose data register during arithmetic and logical operations.
2. Some instructions are optimised to work faster with the accumulator.
3. In some instructions (MUL and DIV), the accumulator is assumed. No need to write it explicitly.
4. It can be accessed in various ways as 8,16, or 32-bit chunks, referred to as AL, AH, AX, EAX.

Examples:

`MOV EAX,1234H`	Load constant value 4660 into 32-bit accumulator.
`INC EAX`	Add 1 to accumulator value.
`MOV maxval,EAX`	Store accumulator value to memory variable 'maxval'.
`DIV CX`	Divide accumulator by value in 16-bit register CX.

EBX: The Base register.

The Base register can hold addresses to point to the base of data structures, such as arrays in memory.

Examples:

`LEA EBX,marks`	Initialise EBX with address of the variable 'marks'.
`MOV AL,[EBX]`	Get 1-byte value into AL using EBX as the memory pointer.

ECX: The Count register

It is used as a counter in loops.

Examples:

`MOV ECX,100`	Initialise ECX as the FOR-loop index.
……	
`for1:`	Symbolic address label.
……	
`LOOP for1`	Decrement ECX, test for zero, JMP back if non zero.

EIP: The Instruction Pointer

The Instruction Pointer (Program Counter) holds the address of the next instruction.

Examples:

`JMP mylabel`	Forces a new address into EIP.

Further registers: ESI, EDI

ESI:

The Source Index register is used as a pointer for string or array operations.

EDI:

The Destination Index register is used as a pointer for string or array operations.

Examples:

`MOV AX,[EBX+ESI]`	Get one 16-bit word using Base address and Index register.
`MOV EAX,[ESI]`	Moves one 32-bit word from source to destination.
`MOV [EDI],EAX`	

Further registers: ESP, EBP

EBP:

The Stack Base Pointer.

ESP:

The Stack Pointer.

9. CPU Registers, flags, stack (CPU 寄存器，标志，栈)

CPU status flags

EFLAG: The Flag register holds the CPU status flags

1. The status flags are separate bits in EFLAG where information on important arising conditions such as overflow or carry bits, is recorded.
2. A way of communication between one instruction and the subsequent instructions.

Most often used flags:

1. S: sign.
2. Z: zero, result being zero.
3. C: carry, indicates an arithmetic carry.
4. O: arithmetic overflow error.
   - The addition of two large positive numbers which may give a negative result in a Two's complement system.

Example:

```
  0111011111111111
+ 0110111101110111
  1110011101110110
```

OF will be set.

How CPU flags are operated

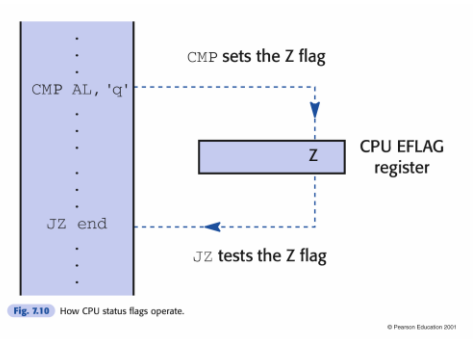


Fig. 7.10 How CPU status flags operate.

CMP: Compare two values, subtract with no result, only setting flags.
JZ: Conditional jump according to the Z flag.

Inline Assembler (内嵌汇编)

1. For the practical sessions we use inline assembler within **Microsoft Visual C++**.
2. Inline means you can insert assembly codes directly into C/C++ programs, compile and execute them.
3. Inline assembly code can refer to C or C++ variables by name:

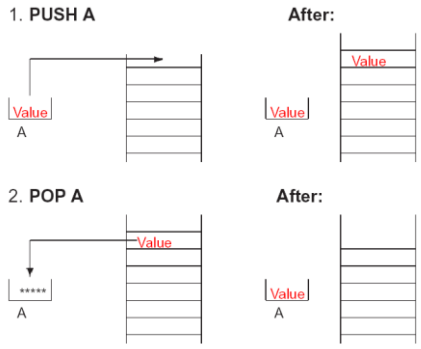
```
_asm mov eax,var      ; stores the value of
                      ; var in EAX
```

a) One can use **_asm** without brackets, in that case it works only for one line.
b) Semicolon is used for the comments in the assembly program. Alternatively, one can use // for the comments.

4. An `_asm {……}` block can call C functions, including C library routines.
   - We use C library routines scanf and printf for input and output in our programs.

Stack (栈)

1. **Stack** is a memory arrangement (data structure) for storing and retrieval information (values).
2. The order of storing and retrieving the values for the stack can be described as **LIFO (last in, first out) (后进先出)**.
3. The **ESP** register stores the address of the item that is on top of the stack.
4. Example of an alternative memory arrangement is a **queue (队列)**: **FIFO (first in, first out) (先进先出)**.
5. Every stack is equipped with two operations: **Push and Pop (入栈和出栈)**.



Acceptance:

1. Stack is a simple but useful data structure.
2. Almost any assembly language has special instructions for implementing a stack.
   - In inline assembly language there are PUSH and POP instructions.
     a) PUSH and POP operations use the stack pointer
     b) register ESP to hold the address of the item on top of the stack.

Using PUSH and POP

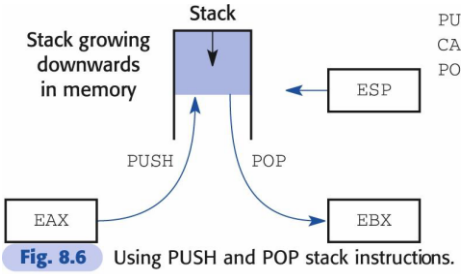
Assume an upside-down stack in memory.

PUSH instruction works:

1. First by decrementing the address in ESP.
2. Then using the ESP address to write the item into stack.

POP instruction works:

1. The item is read using ESP address.
2. The ESP address is incremented by a correct amount to remove the item from the stack.




```
PUSH EAX  ;push 32bit word in A into stack
CALL print;do something
POP EBX   ;pop the 32bit word from stack
```

Fig. 8.6 Using PUSH and POP stack instructions.

Adjusting stack pointer

Sometimes the stack pointer has to be adjusted explicitly by the programmer.

```
ADD ESP,4    ; take 4 bytes off the stack.
             ; (see example in the Hello World program).
SUB ESP,256 ; create 256 bytes of stack.
```

Stack as the temporary store

```
/*demonstration of the use of asm instructions within a C program*/
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    char format[] = "Hello World\n";
                              // declare variables in C
    _asm
    {
        mov ecx,10            // initialize loop counter
    Lj: push ecx              // loop count index saved on stack
        lea eax,format        // load the address of the array
        push eax              // address of string, stack parameter
        call printf           // use library code subroutine
        add esp,4             // clean 4-byte parameter off stack
        pop ecx               // restore loop counter ready for test
        loop Lj               // dec ECX, jmp back if ECX nonzero
    }                         // back to C
    return 0;
}
```

1. In the above example stack was used as a 'scratch pad' temporary store to keep the value of loop counter and free ECX register for other use.
2. That is a common use of the stack as a temporary store for variables (values).

(注：以栈储存 ECX 中的值，是因为 printf 函数会读写 ECX 中的值)

Passing parameters (传递参数):

printf routine needs extra information on what to print and how to print. It should be passed as parameters. It is done via stack:

1. **`push eax`**
   places the address of the string to be printed on the stack.
2. **`call printf`**
   reads the address from the top of the stack and prints the string, but it does not remove this address from the stack.
3. **`add esp,4`**
   cleans the top of stack (remove address).

10. Structure of instructions, addressing mode

(指令结构，寻址方式)

Structure of instructions

The binary codes of almost all instruction contain three pieces of information:

1. The action or operation of the instruction.
2. The operands involved (where to find the information to operate with).
3. Where the result is to go.

Machine instructions are encoded with **distinct bit fields in the prefix** to contain information about:

1. The operation required.
2. The location of operands and results.
3. The data type of the operand.

Pentium instructions can be from 1 to 15 bytes long, depending on:

1. The operation required.
2. The addressing modes employed.

Addressing modes

1. The way of forming operand addresses.
2. More technically, the way one can compute an effective address
   - Effective address is the address of operand used by instruction.
3. Offering various addressing modes support better the needs of HLLs when they need to manipulate large data structures.

1. Immediate (operand) mode:

Example:

```
mov eax,104
```

- Part of the binary code here is the value (=104) of the operand.

2. Data Register Direct:

Example:

```
mov eax,ebx
```

- The operand is contained in registers and the instruction contains codes of the registers.
- This is the fastest to execute.

3. Memory Direct:

Example:

```
mov eax,a
```

- Here 'a' is a variable, stored in memory and the instruction contains the address of this variable.
- Decode the address into a temporary register within the CPU, and then go to the memory location to read the data into the CPU.

4. Address Register Direct:

Example:

```
lea eax,message1
```

- The instruction, contains the address of 'message1' variable, which is loaded into eax register after the execution of the instruction.
- Commonly used to initialise pointers which can then reference data arrays.

5. Register Indirect:

Example:

```
mov eax,[ebx]
```

- The instruction copies to the eax register the content of a memory location with the address stored in ebx.

Register Indirect mode in Arrays

```
/*Handling arrays with Register Indirect addressing mode*/
    …
    int myarray[5];   // declaration of an array of integers
    myarray[0] = 1;   //
    myarray[1] = 3;   //
    myarray[2] = 5;   // initialise the array
    myarray[3] = 7;   //
    myarray[4] = 9;   //
    _asm
    {
        …
        lea ebx,myarray
        // address of the array (its 0th element) is saved in ebx
        mov ecx,5
        // size of the array is saved in the counter
        mov eax,0
        // eax will be used to store the sum, initialise to 0
loop1:  add eax,[ebx]
        // read an element of the array at the address stored in ebx
        add ebx,4
        // int occupies 4 bytes (32 bits)
        // so to read next element increase the address in ebx by 4
        loop loop1
        // end of the loop
    }
```

6. Indexed Register Indirect with displacement:

Examples:

```
mov eax,[table+esi]
mov eax,table[esi]
```

- In both cases the effective address is obtained by adding the address 'table' contained in the operand field of the instruction to the content of a register (registers).
- The symbol 'table' is the address of the array base.

Alternative array handling

```
/*Handling arrays with Register Indirect with Displacement
addressing mode*/
    …
    int myarray[5];   //declaration of an array of integers
    myarray[0] = 1;   //
    myarray[1] = 3;   //
    myarray[2] = 5;   // initialise the array
    myarray[3] = 7;   //
    myarray[4] = 9;   //
    _asm
    {
        mov ecx,5
        // size of the array is saved in the counter
        mov eax,0
        // eax will be used to store the sum, initialise to 0
        mov ebx,0
loop1:  add eax,myarray[ebx]
        //read an element of the array at the address myarray + ebx
        add ebx,4
        // int occupies 4 bytes (32 bits), so to read next element
        // increase the address in ebx by 4
        loop loop1
        // end of the loop loop1
    }
```

11. Inline assembly, jump, jump on flag

(内嵌汇编，跳跃指令)

Output in inline assembly (内嵌汇编中的输出)

```
…
char format[] = "Hello World\n"; // declare variables in C
…
    …
    lea eax,format
    // load address of the string 'format' into eax
    push eax
    // address of string, stack parameter
    call printf
    // use library code subroutine
    add esp,4
    // clean 4-byte parameter off stack
```

We have seen the above fragment implementing the call to printf function.
Equivalent C code is:

```
printf("Hello World\n");
```

Printing numbers

Print the value of the integer variable 'myint'

C++:

```
printf("%d", myint);
```

1. Qualifier "%d" means the content of 'myint' will be printed as a decimal integer.
2. There are more qualifiers (types) which can be used in printf:
   a) %c print a character
   b) %d or %i print a signed decimal number
   c) %s print a string of characters

Assembly:

1. Push the second parameter (integer variable to the stack).
2. Push the first parameter (actually, address of the string) to the stack.
3. Call printf routine.
4. Clean up top two positions in the stack.

```
/*demonstration of the use of asm instructions within C prog*/
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    char format[] = "%d\n";  // declaration of the format string to
                             // be used in printf function as the //
                             first parameter
```

```
    int myint = 157;         // declaration of an integer variable
    _asm
    {
        push myint
        // push the value of the variable onto the stack
        lea eax,format
        // address of the format string is saved in eax
        push eax
        // push the address of the string to the stack
        call printf
        // call printf, it will take two parameters from the stack
        add esp,8
        //clean up top two positions in the stack
    }
    return 0;
}
```

Input in inline assembly (内嵌汇编中的输入)

Input the value into the integer variable 'input'

C++:

```
scanf("%d", &input);
```

1. Qualifier "%d" means the input will be read as a decimal integer.
2. &input presents the address of the variable 'input'.
3. As for printf there are more qualifiers which can use in scanf:
   a) %c read a single character
   b) %d read a signed decimal integer
   c) %s read a string of characters until a white space or terminator (blank, new line, tab) is found

Assembly:

```
…
char format[] = "%d";
int input;    //declaration of an integer variable for user's input
_asm
{
    lea eax,input
    push eax
    lea eax,format
    //address of the format string is saved in eax
    push eax
    // push the address of the string to the stack
```

```
    call scanf
    // call scanf, it will take two parameters from the stack
    // scanf(%d,&input);
    // user's input will be put in the 'input' variable
    add esp,8
    // clean top two positions in the stack
}
…
```

Controlling program flow (控制程序流程)

1. Very few programs execute all instructions sequentially, from the first till the last one.
2. Usually, one needs to control the flow of the program
   a) Jump from one point to another, often depending on some conditions.
   b) Repeat some actions while some condition is maintained, or until some condition is reached.
   c) Passing control to and from procedures.

Jumps

Jump is the most straightforward way to change program control from one location to another.

Jump instructions fall into two categories:

1. Unconditional.
2. Conditional.

Unconditional jumps (无条件跳跃)

JMP instruction transfers control unconditionally to another instruction.

It has a syntax:

```
JMP <address of the target instruction>
```

.

The address of the target instruction is given by its label.

Typical use of JMP instruction:

```
label1:     …
            …
            jmp label1
label2:     …
            …
            jmp label2
            …
keep going: …
```

1. 'label1', 'label2' and 'keep going' are all labels.
2. Unconditional jumps skip over code that should not be executed.

Conditional jumps (条件跳跃)

Conditional jumps work as follows:

1. First test the condition.
2. Then jump if the condition is true or continue if it is false.

There are more than 30 jump instructions:

1. Two of them, JCXZ and JECXZ, test whether the counter register CX, or ECX is zero.
2. Remaining jump instructions test the status flags.

Example:

```
          …
          …
          jecxz finish
          mov eax,inp
          …
          …
finish:   add esp,4
```

When 'jecxz finish' instruction is executed:

If ecx register contains 0, then the next instruction to execute is

```
add esp,4
```

Otherwise, the next instruction to execute is

```
mov eax,inp
```

Jumping based on status flags

Instruction	Jump if
JC/JB	Carry flag is set (=1)
JNC/JNB	Carry flag is clear (=0)
JE/JZ	Zero flag is set (=1)
JNE/JNZ	Zero flag is clear (=0)
JS	Sign flag is set (=1)
JNS	Sign flag is clear (=0)

JO	Overflow flag is set (=1)
JNO	Overflow flag is clear (=0)

Jumps Based on Comparison of Two Values:

The CMP instruction is the most common way to test for conditional jumps.

It compares two values without changing them, while it changes the status flags according to the results of the comparison.

Example:

if the value of eax and ebx are the same, then the execution

```
cmp eax,ebx
```

will set zero flag Z=1.

Jumps Based on Comparison:

Instruction	Jumps if (assuming execution just after CMP)
JE	The first operand (in CMP) is equal to the second operand.
JNE	The first and second operands are not equal.
JG	First operand is greater.
JLE	First operand is less or equal.
JL	First operand is less.
JGE	First operand is greater or equal.

Example:

```
cmp ax,bx    ; Compare AX and BX
jg label1    ; Equivalent to: If (AX > BX) go to label1
jl label2    ; Equivalent to: If (AX < BX) go to label2
```

```
add ax,input    ; Add input to AX
cmp ax,0        ; Compare AX with 0
jge label1      ; Equivalent to: If (AX >= 0) go to label1
jl label2       ; Equivalent to: If (AX < 0) go to label2
```

12. Controlling program flow, loop (控制程序流程，循环)

Loops (循环)

Simple loop:

```
loop <label>
```

Automatically decrements ECX
When ECX=0, the loop ends
When ECX is not 0 it jumps to <label>

Example:

Repeat by counting down from 200 to 0 and do some task in the loop.

```
        mov ecx, 200    ; set counter
next:   …               ; do the task here
        …
        loop next       ; jump to the label 'next'
        …               ; continue after looping
```

Loop upon two conditions

LOOPNE instruction
While EAX is not equal to EBX **AND** not 200 times yet

```
        mov ecx,200     ; Set counter
next:   …               ; Set label
        …               ; Do something
        cmp eax,ebx     ; Are EAX and EBX the
                        ; same? Or 200 times already?
        loopne next     ; No? Go to
        …               ; Yes?
```

Conditional statements (条件语句)

Java:

```
if (c > 0)
pos = pos + c;
else
neg = neg + c;
```

Assembly code:

1.

```
                mov eax,c
                cmp eax,0
                jg positive
negative:       add neg,eax
                jmp endif
positive:       add pos,eax
endif:          …
```

2.
```
            mov eax,c
            cmp eax,0
            jle negative
positive:   add pos,eax
            jmp endif
negative:   add neg,eax
endif:      …
```

The for statement (for 语句)

Java:

```
for (int x = 0; x < 10; x++)
{
   y = y + x;
}
```

Assembly code:

```
            mov eax,0
for_loop:   add y,eax
            inc eax
            cmp eax,10
            jl for_loop
```

The while statement (while 语句)

Java:

```
while (fib2 < 1000)
{
   fib0 = fib1;
   fib1 = fib2;
   fib2 = fib1 + fib0;
}
```

Assembly code:

```
while:      mov eax,fib2
            cmp eax,1000
            jge end_while
            mov eax,fib1
            mov fib0,eax
            mov eax,fib2
            mov fib1,eax
            add eax,fib0
            mov fib2,eax
```

```
            jmp while
end_while:  …
```

Switch Case statement (switch case 语句)

Java:

```
switch (num){
   case 1: … ;
      break;
   case 2: … ;
      break;
}
```

Assembly code:

```
            mov eax,num
            cmp eax,1
            je case_1
            cmp eax,2
            je case_2
            jmp end_case
case_1:     …
            jmp end_case
case_2:     …
end_case:   …
```

Equivalent without loop construction:

```
      mov ecx, 200
next: …
      …
      loop next
```

```
      mov ecx,200
next: …
      …
      dec ecx
      cmp ecx,0
      jne next
```

13. Subroutines, call, return (子程序，调用，返回)

Subroutines

Definition:

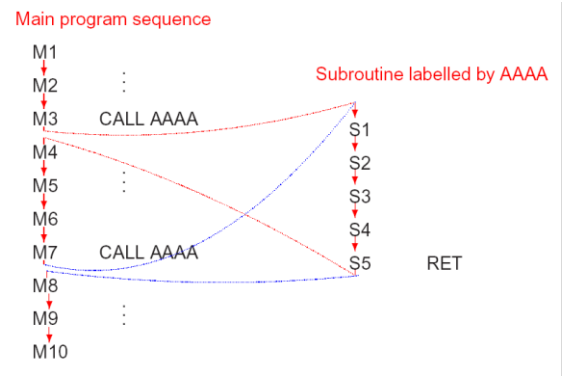
1. A subroutine is a general term. In different programming languages it may be called differently:
   a) Procedure in Pascal.
   b) Function in C.
2. Yet the idea is the same: a subroutine is a part of the code, which can be used repeatedly within the program that is being executed.
3. The subroutine is invoked at different moments from two different places in the main program.

Advantages:

1. Save the effort in programming.
2. Reduce the size of programs.
3. Share the code.
4. Encapsulate, or package, a particular activity.
5. Hide complications from the user.
6. Provide easy access to tried and tested code.
7. Facilitate code reuse.

Subroutines in assembly language

(注：VC++内嵌汇编不支持定义过程，以下涉及 PROC 和 ENDP 的代码均无法在 Visual Studio 2019 VC++环境下运行)



Return address:

1. Different calls of the same subroutine return to different places.
2. One needs to store a return address for every call of the subroutine.

Example:

Declaration of a simple procedure looks as follows:

```
label PROC
      ...
      ...
      RET       ; return
label ENDP
```

The procedure can be called by the instruction:

```
call label
```

The instruction RET changes the control, causing execution to continue from the point following the CALL instruction.

CALL

CALL does the following:

1. Records the current value of EIP (Instruction Pointer) as the return address.
2. Places the required subroutine address into EIP (instruction Pointer), so the next instruction to execute is the first instruction of the subroutine.

Nested calls: (嵌套调用)

```
        . . .
        :
        CALL SUB1       * call first subroutine
        :
SUB1:   . . .
        :
        CALL SUB2       *call second subroutine
        :
        RET
SUB2:   . . .
        :
        RET
```

If the return address was stored in a dedicated memory location, it would work for the first call. But when it comes to the second call, we would have no place to store the return address. In the last example we need both return addresses stored at the same time. The more deep nesting of subroutine calls, the more space do we need to store all return addresses.

An elegant solution is to **use a stack to store all return addresses**.

CALL and RET working together

CALL does the following:

1. Records the current value of EIP (Instruction Pointer) as the return address: copy the address from EIP to the stack (PUSH).
2. Puts the required subroutine address into EIP (Instruction Pointer), so the next instruction to execute is the first instruction of the subroutine.

RET does the following:

1. Pop the last address stored in the stack and put it into EIP.

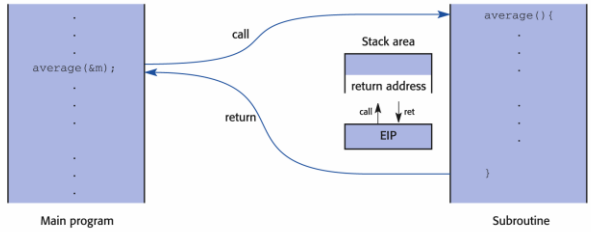


Fig. 8.3 Using the stack to hold the subroutine return address.

14. Parameters passing, ref parameter, value Parameter

(参数传递，引用传递，按值传递)

Parameters

In general, parameters can take a number of forms, depending on the nature of the information required by the subroutine.

Two main forms of parameters:

1. Value parameters.
2. Reference parameters.

Value parameters (按值传递)

In many cases, the additional information required by a subroutine will be a simple value (or values) of some type(s):

- For example, a numeric value, or ASCII code value, etc.

Such parameters are called value parameters.

Example:

```
            mov eax, first    ;
            mov ebx, second   ;
            call bigger       ; calling
            mov max, eax      ;
bigger      proc              ; procedure which uses value parameters
            mov save1, eax    ; passed through registers EAX and EBX
            mov save2, ebx    ;
            sub eax, ebx      ; the result of the procedure is
                              ; returned again via register EAX
            jg first_big      ;
            mov eax, save2    ;
            ret               ;
first_big:  mov eax, save1    ;
            ret               ;
bigger      endp              ;
```

Reference parameters (引用传递)

A variable is a location in the memory, the name of the variable determines the address of this location.

Such kind of parameters are called reference parameters.

Example:

```
            lea eax, first    ;
            lea ebx, second   ;
            call swap         ; calling
finish:     …                 ;
swap proc                     ; subroutine which uses
            mov temp,[eax]    ; reference parameters
            mov [eax],[ebx]   ; passed via registers
            mov [ebx],temp    ;
            ret               ;
swap endp                     ;
```

Passing parameters via registers or stack

The simplest method of passing parameters into a subroutine is to use a register:

- Copy a value into an available CPU register and then jump to the subroutine.

Both value and reference parameters can be passed using registers.

But this method would be too constraining!
For subroutines with a few parameters, one may pass these parameters (in address, or value form) using general purpose registers
However, in general, the number of registers available is very limited, and is not enough to implement subroutines with an arbitrarily large number of parameters.
Thus, we need to use the memory in order to implement parameter handling for subroutines.
The solution in most computers is to use the stack for passing parameters and keeping the local variables.

Stack frame (堆栈结构)

The area of the stack which holds all data related to one subroutine call is called a stack frame.
This data includes:
1. Parameters of the subroutine.
2. Return addresses.
3. Old stack pointer contents (EBP)
4. Local variables.

EBP and ESP for stack frames

Because of the nested calls several stack frames may be present at the same time.
Two registers are used to work with stack frames:
1. EBP: The stack frame base
   - To indicate the base of the current stack frame.
   - EBP initially contains an address of the base of the stack.
2. ESP: The stack pointer.
   - To hold the address of the top of the stack.
   - ESP is always pointing to the top of the stack.

Process:
Just before and during the call of a subroutine the following happens:
1. The parameters are pushed on the stack.
2. The return address is pushed on the stack.
3. The address stored in EBP is pushed on the stack.
4. A new stack frame is created.
5. The current address of the top of the new stack frame is saved in EBP.
6. The local variables are installed on the new stack.

Once the subroutine done its job:
1. Pop all local variables out of the stack.
2. Pop the previous EBP address from the top of the stack and restore it in EBP (stack frame base pointer).
3. Clean up parameters in the stack.
4. Pop the return address and save it in EIP.

(Note: popping order is crucial.)

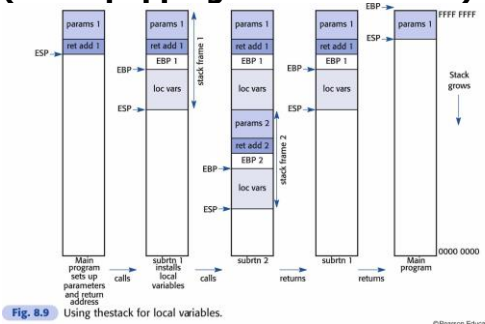


Fig. 8.9 Using thestack for local variables.

15. Recursive subroutines (递归子程序)

Recursive subroutines

1. A recursive subroutine, or procedure, is one that may in some circumstances calls itself to perform some subsidiary task.
   - For example, a subroutine SUBR may CALL SUBR
2. Recursion may appear in a more subtle form of mutual recursion, when, e.g.,
   - SUBR1 calls SUBR2
   - SUBR2 in turn calls SUBR1

Example:

1. Factorial function.
   - factorial(1) = 1.
   - factorial(n) = n * factorial(n-1).
2. Merge sort.
   - given a list, split it into two parts
   - apply Merge sort to each part
   - merge the results

Factorial in the assembly language.

Auxiliary procedure (辅助程序):

```
multiply   PROC          // input from two top values on the stack
           pop eax       // pop a value from the stack to eax
           mov aux eax   // move this value to the auxiliary variable
           pop eax       // pop one more value from the stack
           mul eax,aux   // multiply these two values
           ret           // return the result in edx:eax
multiply   ENDP
```

It takes two top values on the stack, multiplies them and return the result in eax register.

Side effect:

1. Stack modified (2 pops)
2. eax original contents replaced with the product

Main procedure (主程序):

```
factorial  PROC              // input n in eax
           push eax          // push current value onto the stack
           dec eax           // decrease the value of n
           jz finish         // if it is zero go to finish
           call factorial    // otherwise call factorial
           push eax          // push the result of last factorial's
                             // call to the stack
           call multiply     // call multiply subroutine
           ret               // return
```

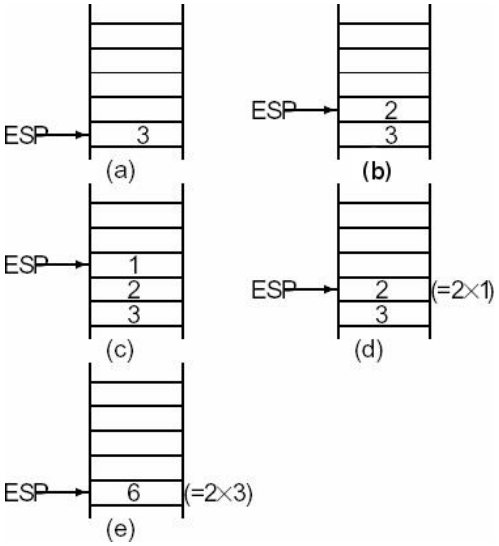
```
finish:    pop eax            // pop the parameter from the stack into
                              // eax
           ret                // return with the result in eax
factorial  ENDP
```

Comments:

1. Parameter n is passed to factorial subroutine via register eax.
2. The register eax is used also to return the result.
3. The 2 parameters to multiply subroutine are passed via stack.
4. The result of multiply is also returned via eax.

Example:

```
mov eax,3
call factorial
```



16. Sign-and-magnitude representation, complements (符号及值表示法，补码)

Numbers vs. text

1. Numbers (consisting of numeric characters) are often processed differently from text. But still they must be entered into computer just like any other alphanumeric characters, one digit at a time.
2. Conversion between alphanumeric representation of numbers and special representation of numbers is done usually in a program (software).
3. To store numbers, we need an encoding scheme, which would allow us to
   a) The algebraic sign of numbers (+/-).
   b) Decimal point that might be associated with a fractional number.

Two categories of integer data

1. Unsigned integers (无符号整数)
2. Signed integers (有符号整数)

Unsigned integers

Unsigned binary representation:

It just stores any whole number in its binary representation.

1. Thus, a byte can store any unsigned integer of value between 0 and 255.
2. A 32-bit word can store unsigned integers in a range $0 - (2^{32} - 1) = 4{,}294{,}967{,}295$.
3. Multiple storage locations (words) can be used to represent bigger unsigned integers.

Binary coded decimal (BCD) (BCD 码或二-十进制编码):

1. Digit by digit binary representation of original decimal integer.
2. Each decimal digit is individually converted to binary.
3. This requires 4 bits per digit (not all 4 bits patterns are used).
4. One byte can store unsigned integers in a range 0-99 under BCD encoding.

Example:

Decimal value 68 is presented in BCD as 01101000.

Binary vs. BCD representation:

1. BCD is less economical than binary representation.
2. Calculations in BCD are more difficult.
3. But, translation between BCD and character form is easier.
   - Last 4 bits of ASCII numeric character codes correspond precisely to the BCD representation.
   - Example: ASCII code of '5' is 0110101, its BCD representation is 0101.
4. Binary representation is more common, BCD is used for some business applications.

Signed integers

Sign and magnitude representation (符号及值表示法):

It is representation of signed integers by a plus or minus sign and a value.

Agreement:

Left most bit represent a sign, e.g., 0 stands for + and 1 stands for -.

Example:

1. 00100111 represents 39.
2. 10100111 represents -39.

Features:

1. Calculations are more difficult than in the case of binary (unsigned) representation.
2. There are two different binary codes for the number 0: 00000000 and 10000000 (8-bit codes).
3. Positive range of the signed integer is one half as large as the unsigned integer of the same number of bits. The same holds for negative range.
4. Thus, 8 bits can represent the numbers from 127 to 127 (0 being represented twice).

Complementary representations (补码表示):

10's complementary coding:

Representation	$500 \ldots$ 999	$0 \ldots 499$
Number being represented	$-500 \ldots -001$	$0 \ldots 499$

Thus, the numbers (3-digit sequences):

1. From 000 to 499 are representing themselves.
2. Any number x from 500 to 999 represents negative number -(1000-x), that is, negated complement to 1000.
3. Number 0 is not represented twice.

Method of complements:

1. A technique in mathematics and computing used to subtract one number from another using only addition of positive numbers.
2. Commonly used in mechanical calculators and in modern computers.

Examples of (x-y):

Assume $x \geq y$,

Instead of subtracting y from x, the complement of y is added to x and the leading '1' of the result is discarded.

$$\begin{array}{r} 373 \quad (x) \\ -\ 218 \quad (y) \\ \hline ? \end{array} \quad \rightarrow \quad \begin{array}{r} 373 \quad (x) \\ +\ 218 \quad (10\text{'s complement of y}) \\ \hline 1155 \end{array}$$

The first '1' digit is then dropped, giving 155, the correct answer.

In the case $x < y$,

There will not be a '1' digit to cross out after the addition.

$$\begin{array}{r} 185 \quad (x) \\ -\ 329 \quad (y) \\ \hline ? \end{array} \quad \rightarrow \quad \begin{array}{r} 185 \quad (x) \\ +\ 671 \quad (10\text{'s complement of} -\text{y}) \\ \hline 865 \quad (10\text{'}s\ complement\ of\ the\ result - 144) \end{array}$$

Representing negative numbers in 10's complements.

17. Addition and Subtraction (加法和减法)

10's complement representation

1. For the 10's complementary representation we have to specify the number of digits (some n).
2. The number being represented depends on that n. For example:
   a) 3-digit complement: '567' represents 433.
   b) 4-digit complement: '0567' represents 567.
3. In 10's complement system the range is **unevenly (不均匀地)** divided between positive and negative integers.
   - For example, 4-digit complement system represents all the integer values between -5000 and 4999.

Overflow testing (溢出检测)

Example:

Codes: '347' + '230' = '577'.

Values: 347 + 230 = 423.

It is overflow.

The result 577 is too big to be fit into 3-digit complement representation (-500 - 499).

Simple rule:

If both inputs to an addition have the same sign, and the output sign is different, overflow has occurred. (如果两个加/减数符号相同，而结果符号与其相异，则发生溢出)

Two's complement

binary codes:

$$10000000 \ldots 11111111 \mid 00000000 \ldots 01111111$$

two's-complement:

$$-128_{10} \quad \ldots \quad -1_{10} \mid 0_{10} \quad \ldots \quad 127_{10}$$

We have to specify the number of digits (here: 8 bits).

1. Numbers that begin with 0 are considered to be positive representing themselves.
2. Numbers that begin with 1 are representing negative numbers.

A complement of the number in 2's complementary n-digit encoding can be found in one of two ways:

1. Either subtract the value from the modulus.
2. Or, invert all bits and add 1 to the result.

Examples:

-11011101 represents -00100011

Consider 2's complementary code 11011101.•

The code started with 1, it follows a negative number is represented.

To find the number:

1. We first invert all bits of code: 11011101 → 00100010.
2. Then add 1, 00100010 + 1 = 00100011.

3. The binary number represented is 00100011.

Numerical types in Java

In Java we have the following numerical data types for integers.

1. byte 8-bit: integers from -2^{8-1} to $2^{8-1}-1$.
2. short 16-bit: integers from -2^{16-1} to $2^{16-1}-1$.
3. int 32-bit: integers from -2^{32-1} to $2^{32-1}-1$.
4. long 64-bit: integers from -2^{64-1} to $2^{64-1}-1$.

2's complement representation of signed integers is used.

18. Floating Point Numbers, IEEE 754

(浮点数，IEEE 754 标准)

Exponential notation (base 10) (基于 10 的科学计数法)

In general, this notation represents numbers in a form $a \times 10^b$.

Components:

1. The sign of the number.
2. The magnitude (量级) of the number, known as the mantissa (尾数)
3. The sign of the exponent (指数)
4. The magnitude of the exponent.
5. The base of the exponent (e.g., 10 or 2).
6. The location of the decimal point (小数点).

Floating point formats

1. Any format for floating point numbers should specify how the components of an exponential notation are stored (in a word, or several words).
2. The base of the exponent and the location of the binary point are standardised as part of the format and, therefore, do not have to be stored at all.

Example:

Suppose, that the standard code consists of space for seven digits and a sign **SEEMMMMM**

So, we have two digits for the exponent and 5 digits for the mantissa.

Trade off: precision (mantissa) vs. range (exponent).

Most commonly, the mantissa is stored using sign magnitude format.

Excess n notation for the exponent:

Excess 50 notation for the 2-digit decimal representation of the exponent:

Representation	$0 \ldots \quad 49 \mid 50 \ldots 99$
Exponent being represented	$-50 \ldots -1 \mid 0 \ldots \quad 49$

Offset the value of the exponent by a chosen amount (here it is 50).

It is simpler to use for exponents than the complementary form.

Result:

Thus, 5-digit excess 50 notation allows us a magnitude range of

$$0.00001 \times 10^{-50} < \ldots < 0.99999 \times 10^{+49}$$

We assume that the decimal point is located at the beginning of five-digit mantissa.

Normalisation of floating point numbers

1. To maximise the precision for a given number of digits, numbers are, usually, stored with no leading zeros.
2. Process of transformation the numbers into such a form is called normalization.

Example:

a) A number: 0.00123×10^7

b) Its normalised form: 1.23×10^4

Floating point in binary

1. Assuming normalised representation one can omit the storage of most significant bit (it is always 1!).
   - So, 23 bits provide 24 bits of precision.
2. Binary point should be specified.
   - Most common choice is after the most significant bit, i.e., 1.
   - Notice, this bit itself is not stored!

Example (excess-128)

Consider the code:

$0\ 10000001\ 11001100000000000000000$

Sign of mantissa is '+' (leftmost bit is 0)

Mantissa is **1.**11001100000000000000000

Exponent is 00000001 (=10000001-10000000)

The number represented is +11.10011000…000

IEEE standard 754

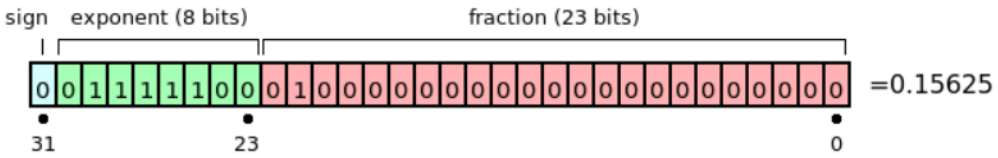
Single precision floating point format (单精度浮点格式)

1. Almost the same format as we have just described, with some exceptions.
2. **32 bits: 1 bit for sign, 8 bits of exponent, 23 of mantissa.**
3. The exponent is formatted using excess-127 notation.
4. Overall, the standard allows approximately 7 decimal digit precision and approximate value range 10^{-45} to 10^{38}

Exponent biasing:

1. The exponent is biased by $2^{8-1} - 1$, that is, biased by 127.
2. Exponents in the range 127 to +127 are representable.
3. e=128 reserved for NaN, infinity.

Example:



The represented number has value v:

$$v = s \times 2^e \times m\text{, where}$$

1. **s = +1 (positive number) when the sign bit is 0.**
2. **s = -1 (negative number) when the sign bit is 1.**
3. **e = exponent 127.**
4. **m = 1.fraction in binary. (The leading '1' is not stored.)**
   Therefore, $1 \leq m < 2$.

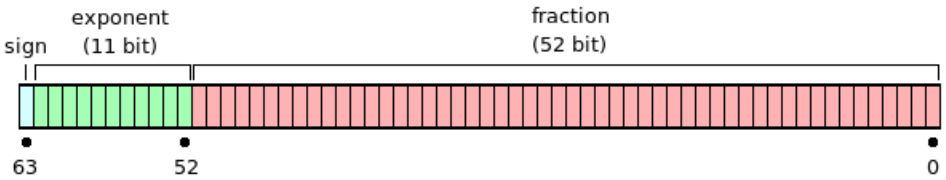
In the above example,
where s = 1, e = -3, m = 1.01 (in binary, which is 1.25 in decimal).
The represented number is therefore $+1.01 \times 2^{-3}$ (in binary), which is +0.15625.

Special cases (e = 128):

1. If exponent is 0 and fraction is 0, the number is 0 (depending on the sign bit).
2. If exponent = $2^8 - 1$ and fraction is 0, the number is infinity (again depending on the sign bit).
3. If exponent = $2^8 - 1$ and fraction is not 0, the number being represented is not a number (NaN).

Double precision floating point format (双精度浮点格式)

1. 64 bits: 1 bit for sign, 11 bits of exponent, 52 of mantissa.
2. The exponent is formatted using excess-1023 notation.
3. Overall, the standard allows approximately 15 decimal digit precision and approximate value range 10^{-324} to 10^{308}.



19. Data storage, RAM, memory (数据存储，RAM，内存)

Data storage

Storage is the capacity of a device to hold and retain data。

Two main types of storage in a computer:

1. Main memory.
2. Mass storage.

Main memory

1. It refers to physical memory that is internal to the computer.
2. The computer can manipulate only data that is inside the main memory.
3. It determines how many programs can be executed at one time and how much space can be allocated to a program.

(注：更多关于主存的细节可在 8. Assembly language and assembler, instruction exec 中找到，这里不再赘述。)

Bytes, Kilobytes, Megabytes, etc

1 Byte (B) = 8 bits.
1 Kilobyte (KB) = 2^{10} bytes = 1024 bytes.
1 Megabyte (MB) = 2^{10} KB = 1024 KB.
1 Gigabyte (GB) = 2^{10} MB = 1024 MB.
1 Terabyte (TB) = 2^{10} GB = 1024 GB.

Typical sizes

1. Typical RAM module has 512 MB / 1 GB of memory.
2. Common size of floppy disk is 1.44 MB.
3. Common size of CD is 650 MB.
4. Common size of memory disc is 1+GB.
5. Common size of DVD is 4.7 GB.
6. Typical hard drive has the size of 120-200-300 GB.

(注：大人，时代变了。本篇（及以下部分篇幅）数据仅供参考。)

RAM

There are two basic types of RAM:

1. Dynamic RAM (DRAM).
   a) Cheaper, but slower.
   b) Implemented via capacitors.
   c) DRAM needs to be refreshed
2. Static RAM (SRAM).
   a) Faster, but more expensive.
   b) Implemented via flip-flops.
   c) No need for refreshing.

Both types of RAM are volatile.

- They lose their contents when the power is turned off.

Refreshing DRAM:

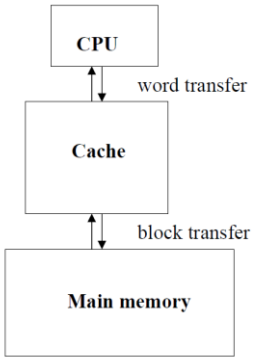
1. Real capacitors leak charge.
2. Stored data eventually fades.
3. To retain data, capacitor charge has to be refreshed periodically.

ROM (Read-only memory)

1. Software stored inside also known as firmware
2. Helps boot up the system
3. BIOS (Basic Input Output System)

Other Forms of Memory

1. Cache memory ("cash")
   a) quick access memory, internal or external to the processor
   b) bridge between the processor and RAM
   c) including **simultaneous read/write**
2. Video memory
   a) VRAM ("vee ram")



Mass storage (大容量存储器)

1. It refers to various techniques and devices for storing a large amount of data.
2. Unlike main memory, mass storage devices retain data even when the computer is turned off.

Types of mass storage:

1. Hard disks.
2. Optical disks: CD ROM, CD RW, DVD, etc.
3. USB disks, Floppy disks.
4. Etc.

Hard Disk Drives (HDD) (硬盘)

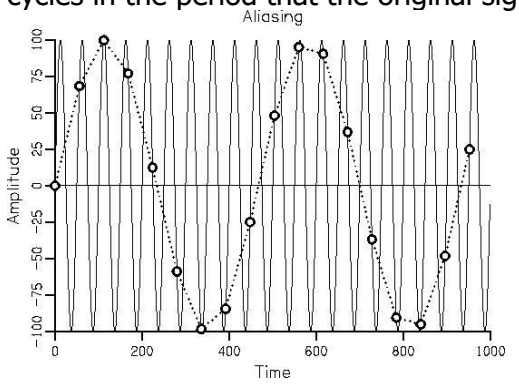
1. Hard disk drives are the most important types of permanent storage used in computers (esp. PCs).
2. Hard disks differ from the other mass storage devices in three ways:
   a) Size (usually larger).
   b) Speed (usually faster).
   c) Permanence (usually fixed in computer and not removable).

RAM vs. Mass Storage (HDD)

1. RAM is volatile, hard disks are not.
2. In general RAM is much faster than mass storage (hard disks):
   a) 10 ns (10^8 s) vs. 10 ms (10^2 s) seek time.
   b) It is about one million times faster.
3. But it is more expensive:
   a) 50 pounds for 8 GB RAM vs. 50 pounds for 1 TB hard disk
   b) 1MB of RAM is about 125 times more expensive.

Analog data → digital data (模拟数据→数字数据)

The solid curve represents the analog signal at a comparatively high frequency. Circles show where samples were taken at a relatively low sampling rate. The dotted line illustrates the apparent frequency of the sampled waveform, completing about two cycles in the period that the original signal completed 20 cycles.



Storage Requirements for Digital Audio

1. CD quality Audio:
   44.1 KHz sampling rate, 16 bits/sample →
   16 bits*44.1KHz = 705,600 bps
   (1KHz = 1000Hz)
2. Stereo:
   44.1 KHz, 32 bits/sample →
   1,411,200bps

20. Memory parameters, memory mapping, cache

(内存参数，内存映射，高速缓冲存储器)

Memory

1. Any memory location in main memory has its own address.
2. It follows then the more memory the larger addresses are needed.

Maximal memory length depends on address width. (最大内存长度取决于地址宽度)

Address width (地址宽度)

Address width is determined by:

1. The number of bits in the CPU address registers such as IP, MAR.
2. The number of lines in the address bus.

Address width	Maximal memory length
16	64 Kbytes
20	1 Mbytes
24	16 Mbytes
32 (Pentium)	4 Gbytes
64 (64-bit architectures)	17 billion Gbytes

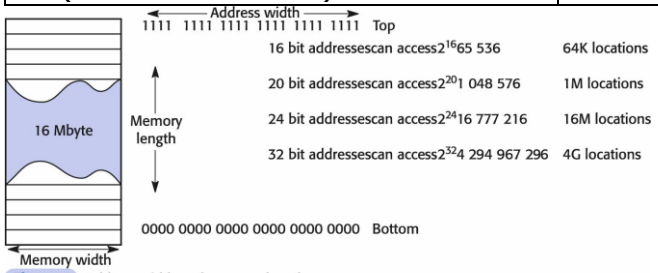


Fig. 3.17 Address width and memory length.

Ideal configuration:

1. With 32-bit address width the maximum memory addressing space is 4 Gbytes.
2. Ideal configuration would be to have a single memory chip and to send an address via 32 address lines.

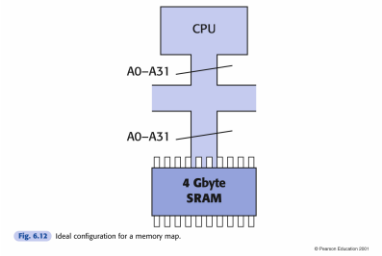


Fig. 6.12 Ideal configuration for a memory map.

Not so ideal in practice

1. 4Gbyte is still a lot of space and many systems do not use it all.
2. Memory chips (DRAM) have a typical size 128 Mbits-256 Mbits-1 Gbits and they are combined into the memory modules of 8-16 chips.
3. For example, the module of 8 of 256 Mbit chips has a capacity of 8 * 256 Mbits = 256 Mbytes.

4. To address memory, we need to select memory chips as well as locations within the chips.

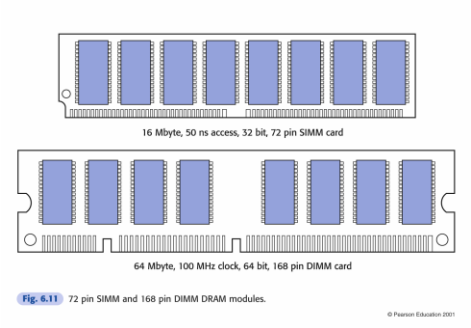


Fig. 6.11 72 pin SIMM and 168 pin DIMM DRAM modules.

Memory mapping (内存映射)

1. A part of the address locates the correct chip.
2. Another part specifies an address within the correct chip.

Memory maps defines as how actually the addresses are mapped to the memory.

Memory address decoding

1. Memory chips are not normally matched to the width of the address bus.
   For example:
   a) CPU may send 32-bit address.
   b) RAM may receive directly 24-bit address.
2. Special Memory Address Decoding circuit implements necessary decoding.

Memory levels

1. Registers.
2. Cache memory (Level 1 and Level 2).
3. Main memory.
4. Mass storage.

Reasons:

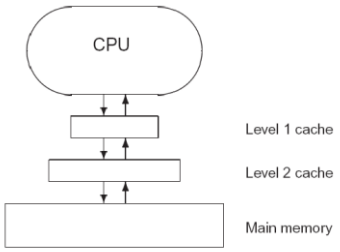
1. To keep up with the demands of faster CPUs.
2. To limit system cost.
3. To cope with ever expanding software systems.

Registers (寄存器):

1. Registers are the memory cells which are core part of the processor itself.
2. It has very fast access (a few nanoseconds).
3. Not that much memory: tens of 32-, 64-, 80-bit registers (typically).

Cache memory (高速缓冲存储器):

1. A memory (more expensive, but faster SRAM) placed between CPU and main memory.
2. Contains a copy of the portion of main memory.
3. The aim is to maintain in fast cache the currently active sections of code and data.
4. Processor when needs some information first checks cache.
5. If not found in cache, the block of memory containing the needed information is moved into the cache.



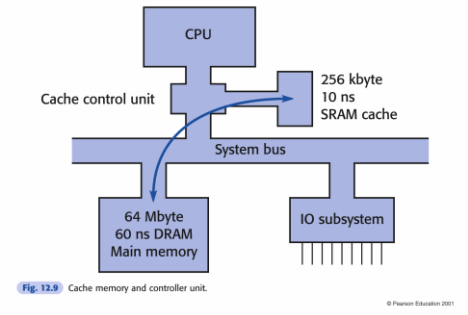
Typically, Level 1 cache has the size 8-64 KB.
Typically, Level 2 cache has the size 128-512KB.

Localisation of access

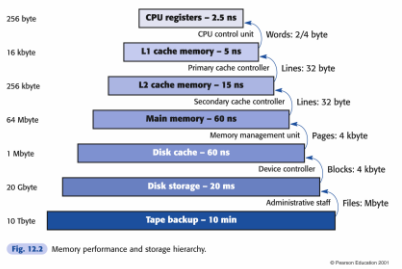
1. The idea of cache memory exploits Localisation of Memory Access principle: Computers tend to spend periods of time accessing the same locality of memory.
2. A portion of code or data which require access needs to be loaded into the fastest memory nearest to CPU.
3. Other sections of the program and data can be held in readiness lower down the memory hierarchy.

Reasons:

1. Partly due to the programmer clustering related data items together in arrays or records.
2. Partly due to the repeating patterns in a program (i.e., loops)
3. Partly due to the compiler attempting to organise the code in an efficient manner.



Memory hierarchy (分级存储器体系)



Going down the hierarchy:

1. Increased capacity.
2. Increased access time.
3. Decreased frequency of access of the memory by the processor.
4. Decreased cost per bit.

21. Mass storage, HDD, Virtual memory

(大容量存储，硬盘，虚拟内存)

Hard Disk Drives (HDD)

1. Hard disk drives are the most important type of permanent storage used in computers (esp. PCs).
2. Hard disks differ from the other mass storage devices in three ways:
   a) Size (usually larger).
   b) Speed (usually faster).
   c) Permanence (usually fixed in computer and not removable).

Schematic diagram:

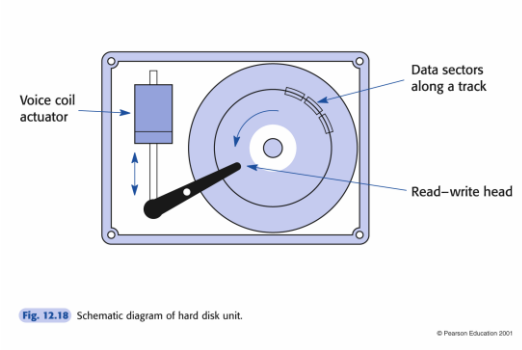
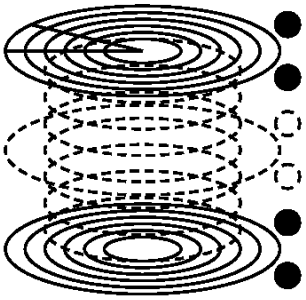
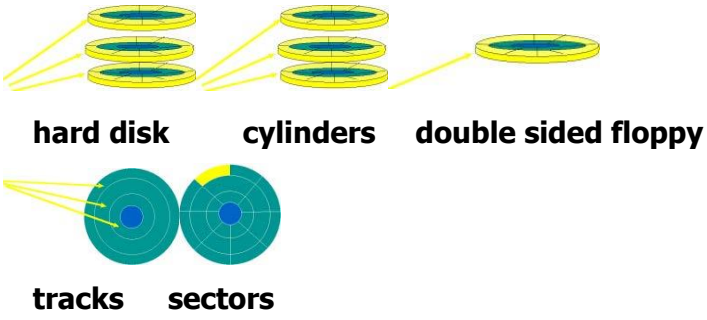


Fig. 12.18 Schematic diagram of hard disk unit.

Components:

1. Each disk **platter (盘片)** has its information recorded on both **surfaces**.
2. Each platter has two **heads (磁头)**.
3. The information is recorded in concentric circles called **tracks (磁道)**.
4. Each track is broken down into smaller pieces called **sectors (扇区)**, each of which holds 512 bytes of information.





Addressing:

1. CHS (柱面-磁头-扇区)
   a) **C**ylinder, **H**ead, **S**ector system.
   b) Telling the disk controller which cylinder, head and sector to access.
   c) Can be mapped onto LBA.
2. LBA (逻辑区块)
   a) **L**ogical **B**lock **A**ddressing, by absolute number of a sector.

Example (simplified):

A piece of information needs to be read.

1. The first step is to figure out where on the disk to look for the needed information.
2. The location on the disk → address expressed
   a) Either: In terms of the cylinder, head and sector (CHS).
   b) Or, in terms of the absolute sector number (LBA).
3. A request is sent to the drive over the disk drive interface giving it this address and asking for the sector to be read.

Hard disk vs. main memory:

1. It is larger.
2. It is slower.
3. It is cheaper (per 1MB).

Disk cache (磁盘缓存)

1. A portion of main memory used as a buffer to temporarily hold data for the disk.
2. Disks write operations are clustered (簇).
3. Some data written out may be needed again. The data are retrieved rapidly from the disk cache instead of slowly from disk.

Storage Technology

1. Retrieving files into RAM is called reading.
   a) loading an application
   b) opening a file
   c) files can be programs or documents
2. Copying data from RAM onto a secondary storage device is called writing.

Files, records, fields, keys:

1. Files
   - e.g., PERSONNEL FILE
2. Records
   - Adam's personal data
   - Adam Smith 35 Manager Purchasing
3. Fields
   - e.g., name, age, position, job function
4. Key
   - e.g., Adam Smith

Virtual memory (虚拟内存)

Virtual memory is a technique, in a sense, opposite to caching:

1. It is the use of low-level memory (i.e., hard disk) to 'expand' high level (main) memory.
2. It provides a convenient expansion of main memory by 'overflowing' data and program code onto magnetic disk.
3. The area on disk reserved for this purpose is known as the swap area.

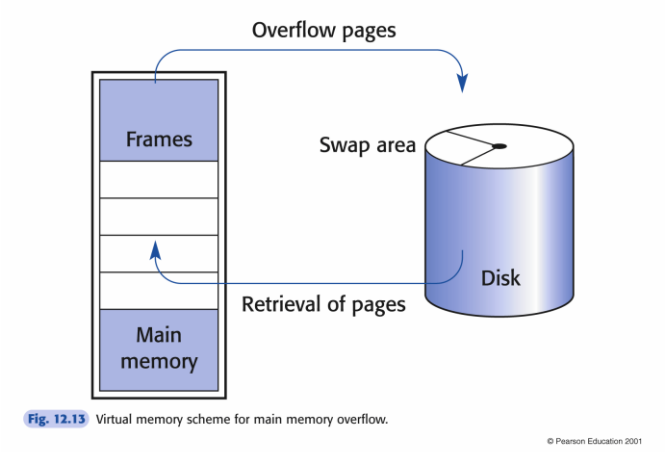
Virtual Memory Management

Features:

1. hard disk space
2. when processor needs more RAM space, swaps unused data onto designated hard disk space.
3. improves flexibility but is slower than RAM to which the processor has direct access.

Components:

1. Main memory is divided into frames, often 4KB.
2. The executable program is similarly divided into frame sized chunks known as pages.
3. When a program is invoked not all the pages are loaded into main memory, only sufficient to get it started.
4. The rest are copied into the disk area, known as swap area.
5. When an instruction is needed from a page not yet in the main memory it is loaded from the disk.
6. If no empty frames exist at the moment the least used frame is freed to allow the new pages to be loaded. This is called swapping.



Virtual Memory Addressing:

1. Within a user program addresses are in a form of 32-bit logical address.
2. In the case of 4KB paging system:
   a) The lower 12 bits are 'address within a page'.
   b) The upper 20 bits serve as the 'page number'.
3. Memory Management Unit maps logical addresses into references to frame numbers and addresses within the frames.

22. Digital electronic circuit, gates (数字电子电路，逻辑门)

Engineering level (技术层面)

1. Engineering model of the computer represents the machine as a complex electrical circuit.
2. Within the circuit there are a large number of physical connections, along each of which a current may flow during the operation of the machine.
   a) Presence of a current is used to represent transmission of the binary digit 1
   b) Absence of a current represents a value 0

Digital Systems (数字系统)

Digital systems based on electronic circuitry

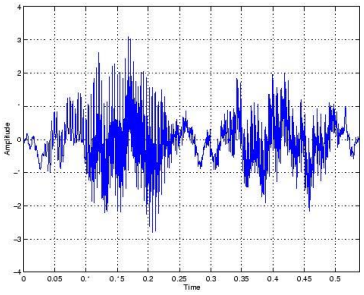
1. 1s and 0s, or on and off
2. Each 1 or 0 is called a bit; or binary digit
3. Computers use digital data representation

Digital electronic circuit (数字电子电路)

This kind of circuit is called digital electronic circuit, because the relevant characteristic is the presence or absence of current (digit 1 or 0), rather than the amount of current flowing.

Analog Systems (模拟系统)

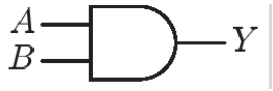
1. continuously variable values, along a range, such as
   a) temperature
   b) pressure values
2. traditional analog recording devices are
   a) humidity recorders
   b) mercury thermometers
   c) pressure gauges
3. standard telephone lines transmit analog signals



Boolean Operations and Boolean Gates (逻辑运算和逻辑门)

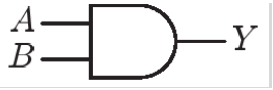
1. All operations that computers perform may be defined in terms of **basic Boolean functions, operating on bits**.
2. The digital electronic circuits and their components can be built from the devices implementing basic boolean operations boolean gates (logic gates).

Boolean Gates:



1. Two of these, labelled A and B, represents electronic inputs to the component, along each of which a current may:
   a) flow (boolean value 1)
   b) not flow (boolean value 0).
2. Third connection, labelled Y represents the output of the component.

AND gate:



1. In the case of AND gate current will flow in Y if and only if a current flow in the input A and in the input B.
2. One may write Y = A and B.
3. Thus, this component implements logic (boolean) operation AND.

Basic logic gates and their truth tables (基础逻辑门和其真值表):

Inputs A B	C A AND B
0 0	0
0 1	0
1 0	0
1 1	1

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

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

Inputs A	NOT A
0	1
1	0

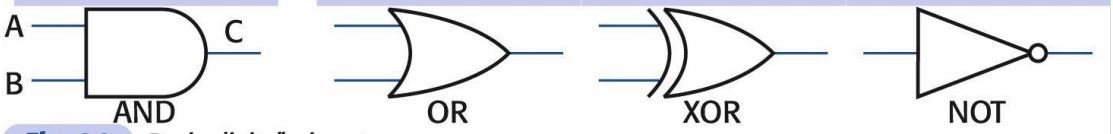


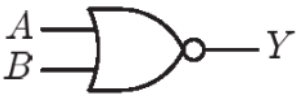
Fig. 4.2 Basic digitallogic gates.

Alternative notations (替代符号):

1. AND: ∧, &
2. OR: ∨
3. XOR: |
4. NOT: ¬, -

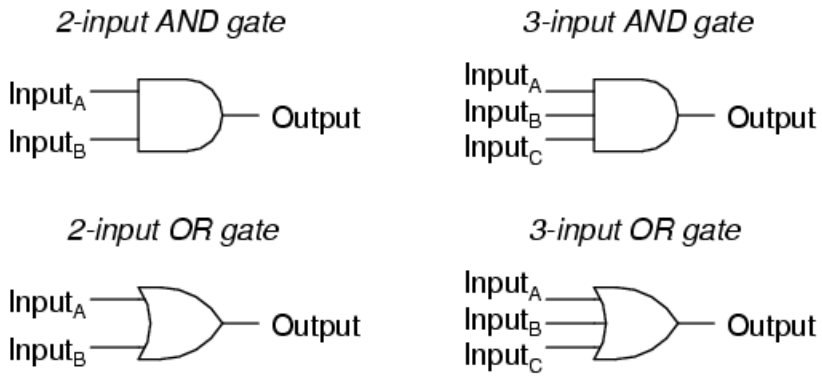
More boolean gates:

NAND gate, Y = not (A and B)



NOR gate, Y = not (A or B)

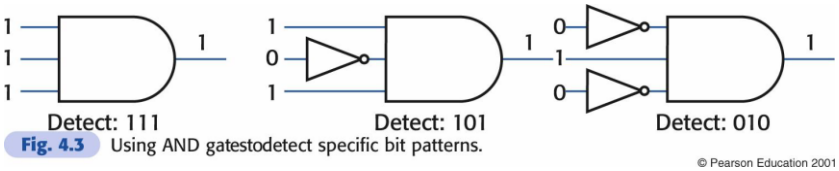
Three input gates:



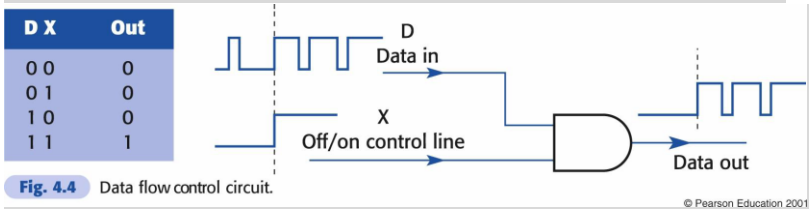
Boolean circuits

1. Elementary boolean gates can be combined into boolean circuits, implementing more complex boolean functions (operations).
2. In fact, any boolean function can be implemented with this set of basic boolean gates.

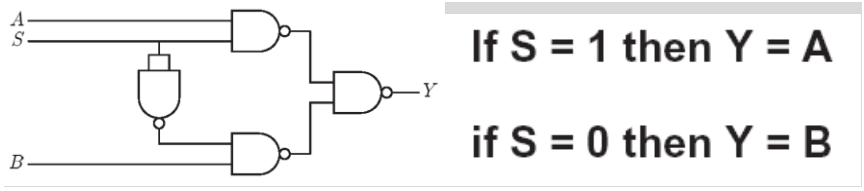
Examples:



Data flow control circuit Filter (数据流控制电路 滤波器)



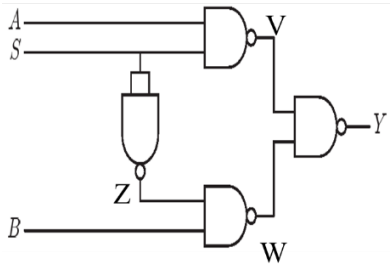
Selector circuit (选择器电路)



23. Data selector, or multiplexer, decoder

(数据选择器，多路器、解码器)

Selector circuit



This circuit implements the function:

1. $Y = not\ (V\ and\ W)$
2. $V = not\ (A\ and\ S)$
3. $W = not\ (Z\ and\ B)$
4. $Z = not\ (S\ and\ S) = not\ S$

Combining altogether we get:

$Y = not\ \big(not\ (A\ and\ S) and\ not\ (not\ (S) and\ B)\big)$

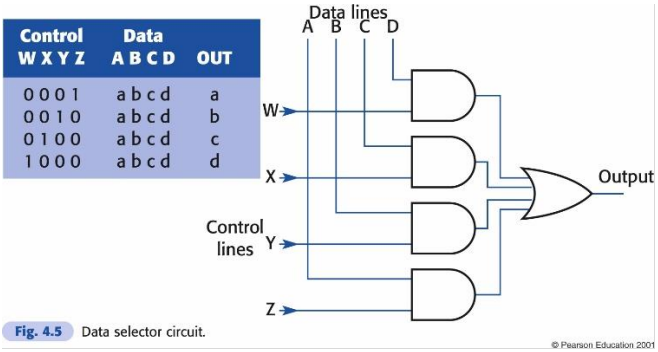
One may give a short definition of the function from the truth table:

If S = 1 then Y = A

if S = 0 then Y = B

$Y = (S\&A) \vee (-S\&B)$

Data selector, or multiplexer



The circuit on the previous slide is a straightforward implementation of the function:

$O = (A\ and\ Z) or\ (B\ and\ Y) or\ (C\ and\ X) or\ (D\ and\ W)$

Disadvantages:

1. The circuit (the function it implements) does not behave as a selector if more than one control line has the signal 1.
2. Use two-line decoder to replace four control lines to solve.

Data selector with two-line decoder:

Selector YX	Line dcba
00	0001
01	0010
10	0100
11	1000

The implementation follows the description:

1. Detect pattern 00 (on YX lines), output the result from line A.
2. Detect pattern 01, output the result from line B.
3. Detect pattern 10, output the result from line C.
4. Detect pattern 11, output the result from line D.

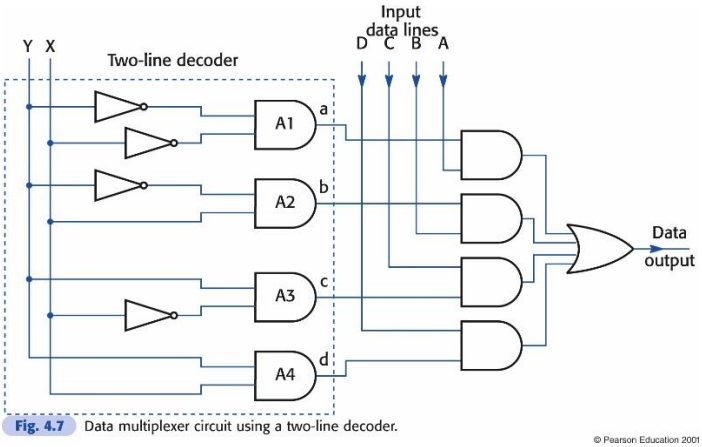


Fig. 4.7 Data multiplexer circuit using a two-line decoder.

Logic equation:

$O = (A\ and\ (not\ X\ and\ not\ Y))$
$or\ (B\ and\ (X\ and\ not\ Y))$
$or\ (C\ and\ (not\ X\ and\ Y))$
$or\ (D\ and\ (X\ and\ Y))$

Cost comparison in gate count:

1. Multiplexer: 4 AND gates + 1 OR gate
2. 2-line decoder: 8 AND gates + 1 OR gate + 4 NOT gates

Implementing a function

Given a truth table for a logic function, to implement the function by a logic circuit one may proceed as follows:

1. Implement detectors (i.e., using AND/NOT gates) for all input patterns on which the function gives the output 1.
2. Connect the outputs of all detectors to the inputs by an OR gate.

Example:

Truth table

i_1	i_2	i_3	i_4	O_1
1	1	0	0	1
1	1	0	1	1
0	0	1	0	1
1	0	1	0	1
0	1	0	1	1

.....	***0***
...	***0***
.....	***0***

Short form

i1	i2	i3	i4	***O1***
1	**1**	**0**	*	***1***
*	**0**	**1**	**0**	***1***
0	**1**	**0**	**1**	***1***

Logic expression:

$O_1 = (i_1\ and\ i_2\ and\ (not\ i_3))$
$or\ ((not\ i_2)\ and\ i_3\ and\ (not\ i_4))$
$or\ ((not\ i_1)\ and\ i_2\ and\ (not\ i_3)\ and\ i_4)$

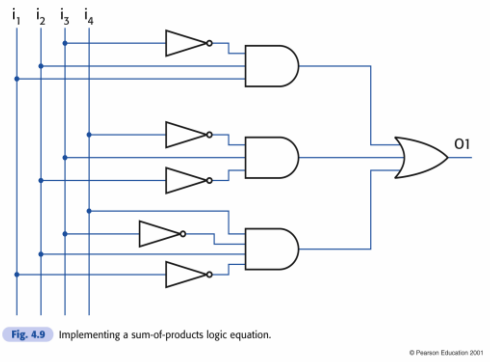


Fig. 4.9 Implementing a sum-of-products logic equation.

24. half adder, full adder, flip-flop (半加器，全加器，触发器)

Half adder

Binary addition. For the addition of single bit binary numbers: (Half-)adder.

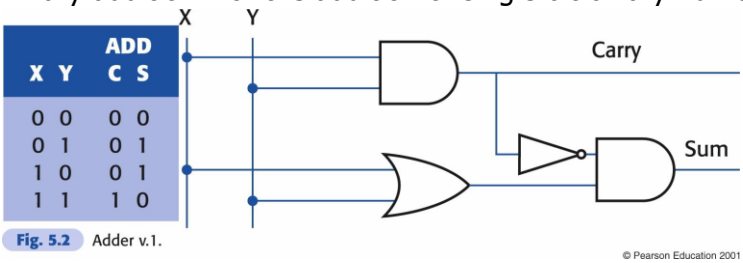


Fig. 5.2 Adder v.1.

Other possible designs for half adder

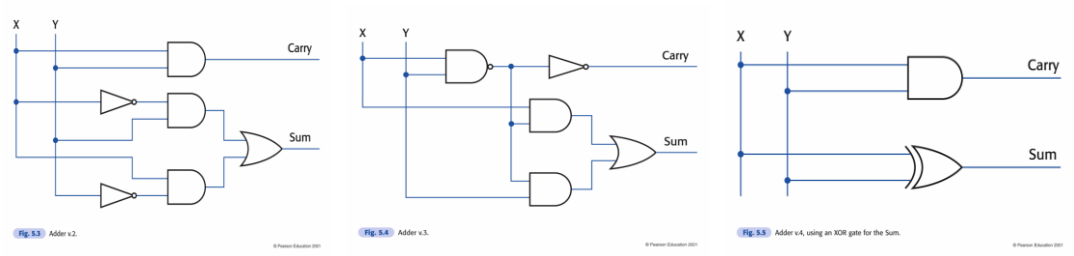


Fig. 5.3 Adder v.2. Fig. 5.4 Adder v.3. Fig. 5.5 Adder v.4, using an XOR gate for the Sum.

Full adder

For the addition of multi bit binary numbers one needs to deal with carry in from previous stage, that means the adder should have three inputs.

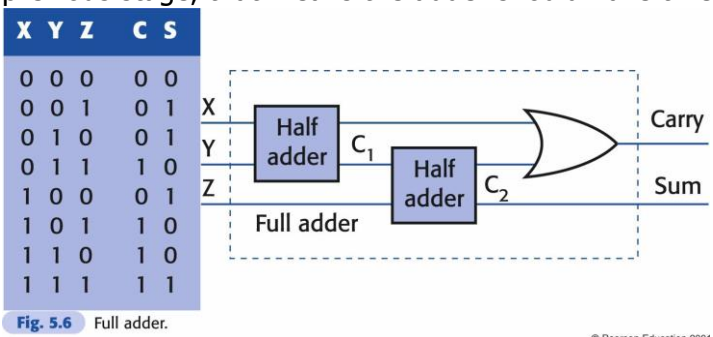


Fig. 5.6 Full adder.

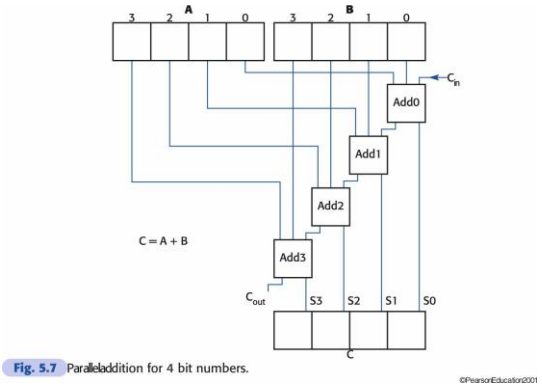


Fig. 5.7 Paralleladdition for 4 bit numbers.

Sequential logic circuits (序逻辑电路)

1. **Combinational** (combinatorial) logic circuits (组合逻辑电路).
   a) In all Boolean circuits that we have seen so far, the output at any moment

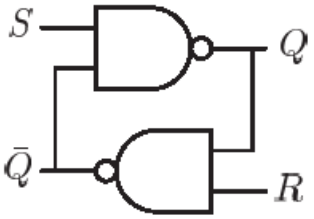
depends only on the input at the same moment.

b) **No memory** is there.

2. **Sequential** logic circuits.
   a) The output depends also on the state of the circuit. The state of the circuit is somehow stored within the circuit.
   b) There is a memory inside.

Flip flops, or latches (触发器)

1. The basic memory element of the sequential circuits is called a flip flop, or latch.
2. The simplest flip flop is made up of two NAND gates. It is called set reset (SR) flip flop.
3. Inputs S are R and while outputs are Q and $\bar{Q}$ (Q 的反相)



- Suppose that S and R are both initially set to 1. Can we determine two outputs Q and $\bar{Q}$? **No.** Two variants are possible:

$Q = 1$ and $\bar{Q} = 0$, or

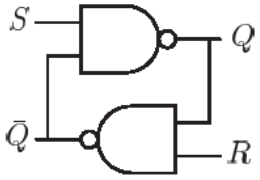
$Q = 0$ and $\bar{Q} = 1$

- Everything is consistent with any of these variants.

1. If we fix both inputs to be 1, then the circuit may have one of the two possible states. The circuit is stable as long as R and S remain at 1.
2. Let's assume that flip flop is in a stable state with Q=1.
   - Suppose, R momentarily becomes 0. This forces changing the value of Q to 0. The circuit will stay in this state even after input R returns to 1. The flip flop switched the states.
   - If then input S momentarily becomes 0 the system will switch the state back.

- Thus, SR flip-flop **remembers** which input was set (momentarily) to 0 last.

- One constraint: the logic surrounding this flip-flop must avoid situations where **both** R and S are 0 at the same time.

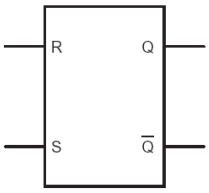


State table of SR flip flop

R	*S*	*Q*
0	**0**	*?*

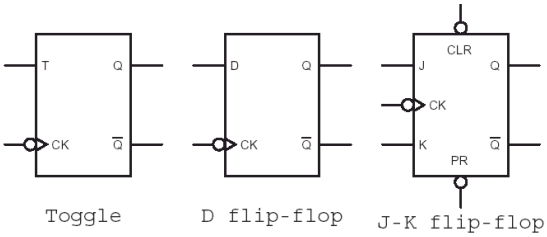
0	**1**	*0*
1	**0**	*1*
1	**1**	Q_{prev}

Representation:



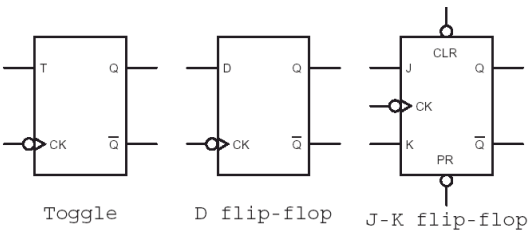
FlipFlop(SR)

Other types of flip flops



Toggle D flip-flop J-K flip-flop

1. All these flip flops has an input marked for clock. The new output occurs when the clock is pulsed, i.e., momentarily changed from 1 to 0.
2. CLR (clear) and PR (preset) inputs are used to initialise the flip flop to known value (0 and 1, respectively).



Toggle D flip-flop J-K flip-flop

T | Q
0 | Q_{prev}
1 | $\bar{Q}_{prev}$

D | Q
0 | 0
1 | 1

J K | Q
0 0 | Q_{prev}
0 1 | 0
1 0 | 1
1 1 | $\bar{Q}_{prev}$

Use of D flip flops Copying data