C & C++ 程序设计

C & C++ 简介

张晓平

数学与统计学院

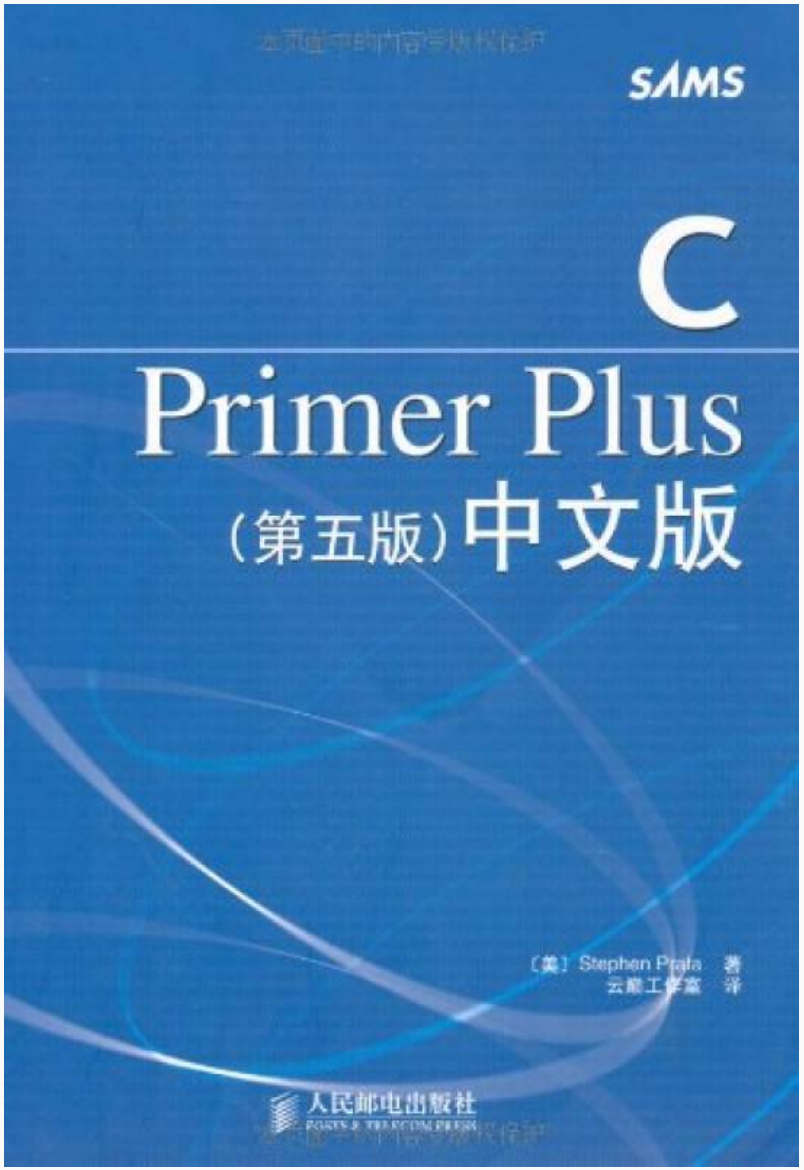
目录


HELLO
World!


```c
#include <stdio.h>
int main(void)
{
  printf("Hello World!\n");
  return 0;
} // helloworld.c
```


```c
#include <stdio.h>
int main(void)
{
  printf("Hello World!\n");
  return 0;
} // helloworld.c
```

```cpp
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World!" << endl;
  return 0;
} // helloworld.cpp
```


SAMS
C
Primer Plus
（第五版）中文版
[美] Stephen Prata 著
人民邮电出版社

计算机的硬件介绍

硬件组成

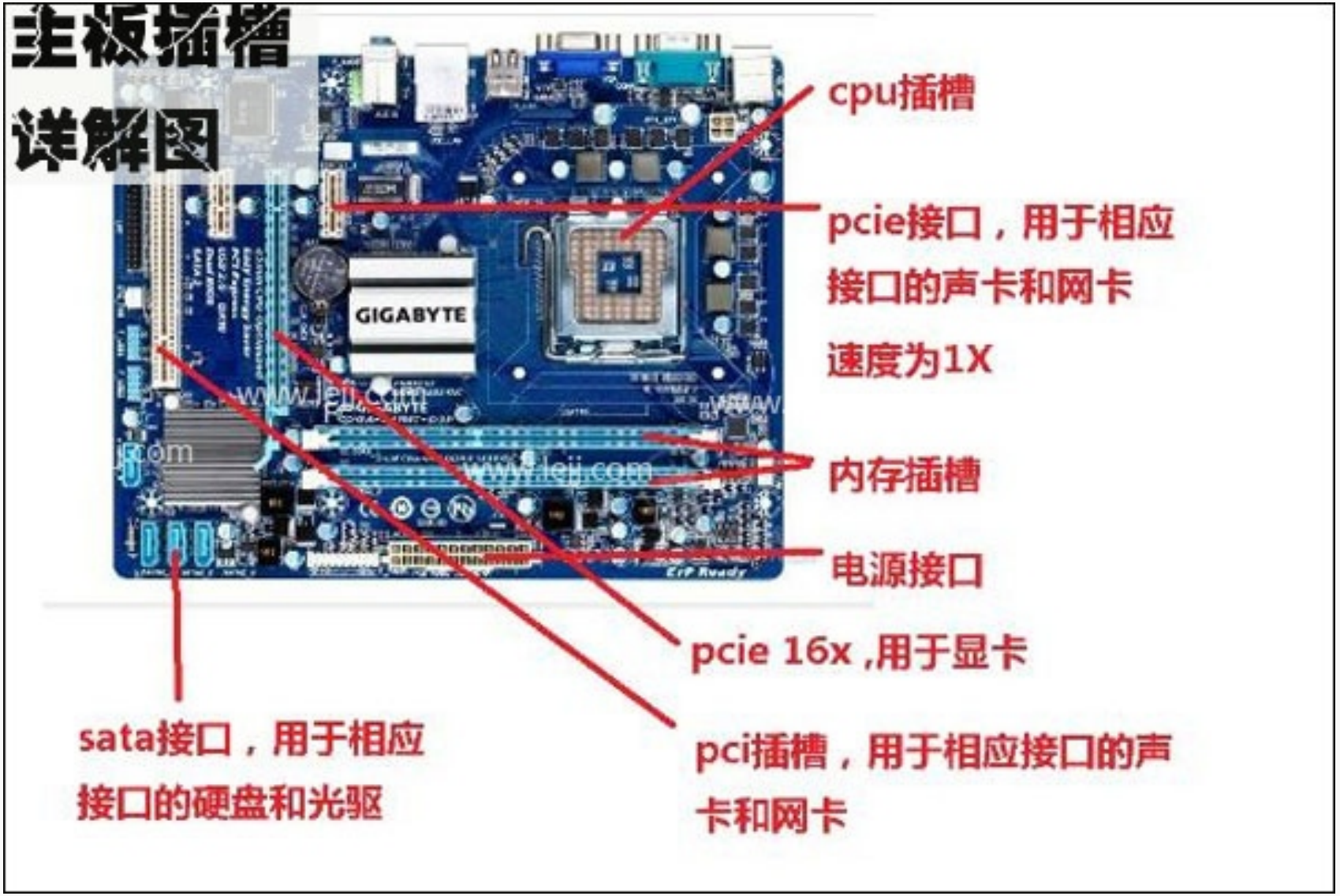
- 中央处理器（Central Processing Unit, CPU）
- 存储器
  - 内存（Memory）
  - 外存：硬盘、U 盘
- 输入设备（Input Device）：键盘、鼠标、摄像头、扫描仪等
- 输出设备（Output Device）：显示器、打印机、音响等

主板

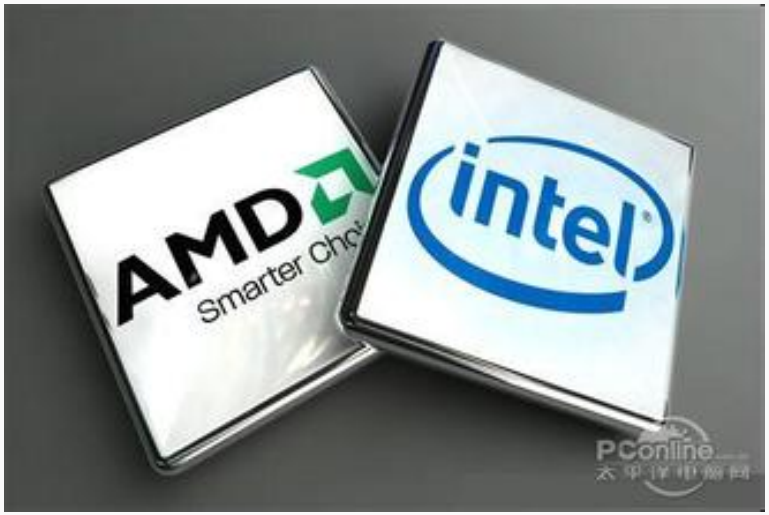
CPU、内存、硬盘、显卡和声卡等都必须安装在主板上才能运行。

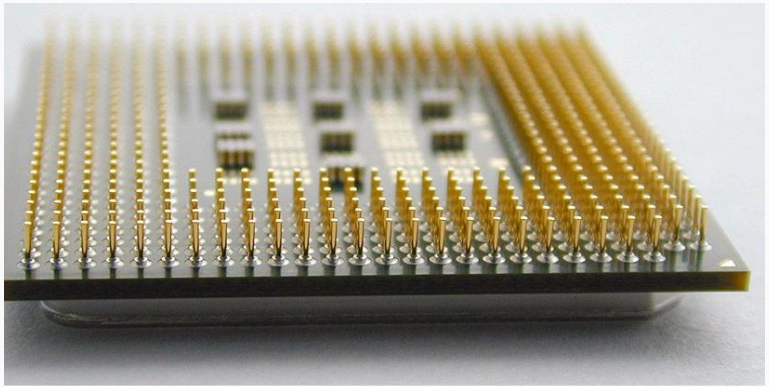
主板是计算机中最大的一块电路板，是计算机系统中的核心部件，它的上面布满了各种插槽（可连接声卡/显卡/MODEM/等）、接口（可连接鼠标/键盘等）、电子元件，它们都有自己的职责，并把各种周边设备紧紧连接在一起。它的性能好坏对计算机的总体指标将产生举足轻重的影响。

主板


主板插槽
详解图
cpu插槽
pcie接口，用于相应
接口的声卡和网卡
速度为1X
内存插槽
电源接口
pcie 16x ,用于显卡
sata接口，用于相应
接口的硬盘和光驱
pci插槽，用于相应接口的声
卡和网卡

CPU





CPU

CPU 是计算机的大脑，计算机处理数据的能力主要取决于 CPU，主要执行以下三种基本操作:

- 读出数据：一般从内存读取数据。
- 处理数据：通过算术逻辑单元对数据进行处理。
- 写入数据：将数据写入内存。

CPU

CPU 由运算器、控制器、寄存器和高速缓冲存储器组成。

CPU

- 算术逻辑单元 (arithmetic logic unit, ALU)：负责对数据进行加工处理，包括
  - 算术运算：加、减、乘、除等
  - 逻辑运算：与、或、非、异或、比较等。

- 控制单元 (control unit, CU)：主要是负责对指令译码，并且发出为完成每条指令所要执行的各个操作的控制信号。

- 寄存器 (register)：来保存指令执行过程中临时存放的寄存器操作数和中间（或最终）的操作结果。
  - 数据寄存器 (Data Rigister, DR)
  - 指令寄存器 (Instruction Register, IR)
  - 程序计数器 (Program Counter, PC)
  - 地址寄存器 (Address Register, AR)
  - 累加寄存器 (Accumulator, AC)
  - 程序状态字寄存器 (Program Status Word, PSW)

内存

内存

- 内存是 CPU 能直接寻址的存储空间，所有程序的运行都是在内存中进行的。
- 只要计算机在运行中，CPU 就会把需要运算的数据调到内存中进行运算，当运算完成后 CPU 再将结果传送出来。
- 其作用是用于暂时存放 CPU 中的运算数据，以及与硬盘等外部存储器交换的数据。

内存

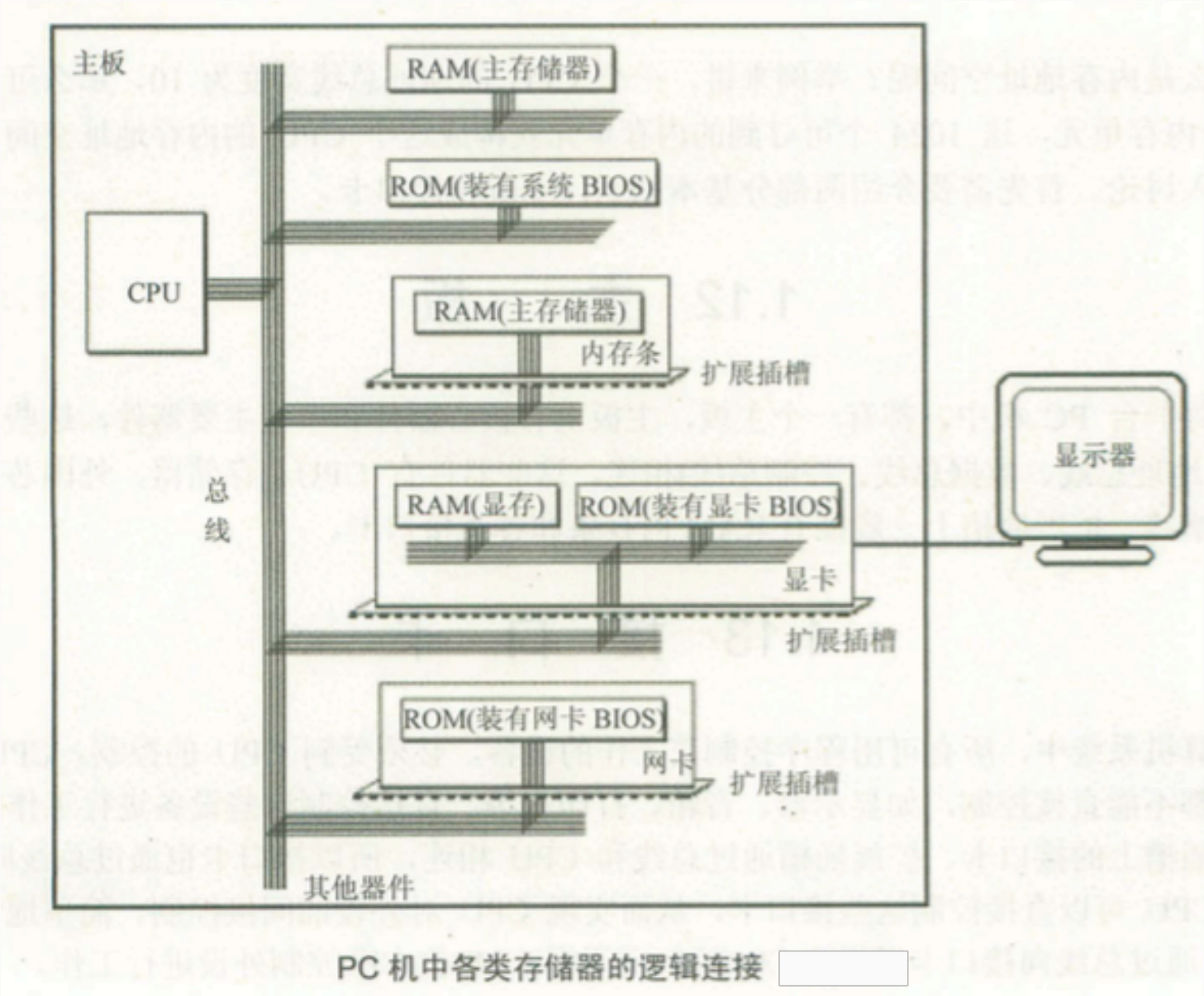
- 只读存储器 (Read Only Memory, ROM)

  只能读取，不能写入，即使断电，存于其中的数据也不会丢失，一般用于存放计算机的基本程序和数据。

- 随机存储器 (Random Access Memory, RAM)

  既可读取，也可写入，断电时，存于其中的数据就会丢失。内存条就是将 RAM 集成块集中在一起的一小块电路板。

各类存储器的逻辑连接



PC 机中各类存储器的逻辑连接

计算机的基本软件组成

计算机的基本软件组成

软件是组成计算机系统的重要部分，分为系统软件和应用软件两大类。

计算机的基本软件组成

- 系统软件是由计算机生产厂商为使用该计算机而提供的基本软件。如操作系统、文字处理程序、计算机语言处理程序、数据库管理程序等。
- 应用软件是指用户为了自己的业务应用而使用系统开发出来的用户软件。如音频视频播放器、QQ、微信等。

系统软件依赖于机器，而应用软件则更接近用户业务。

操作系统（OS）

- 操作系统是最基本也是最重要的系统软件。
- 它负责管理计算机系统的各种硬件资源，如 CPU、内存空间、磁盘空间、外部设备等，并且负责解释用户对机器的管理命令，使它转换为机器实际的操作。
- 常见的操作系统：DOS、WINDOWS、UNIX(LINUX)、OS X 等。

计算机语言处理程序

计算机语言分为机器语言、汇编语言和高级语言。

- 机器语言：机器能直接认识的语言，是由“1”和“0”组成的一组代码指令。
- 汇编语言：实际是由一组与机器语言指令一一对应的符号指令和简单语法组成的。
- 高级语言：比较接近日常用语，对机器依赖性低，即适用于各种机器的计算机语言。如 Basic、Visual Basic、Fortran、C/C++、Java、Python 等。

计算机语言处理程序

将高级语言翻译为机器语言，有两种方式，一种叫“编译”，一种叫“解释”。

- 编译型语言将程序作为一个整体进行处理，编译后与子程序库连接，形成一个完整的可执行程序。
  - 优点：可执行程序运行速度很快
  - 缺点：编译、链接比较费时
  - 常见语言：Fortran、C/C++ 语言
- 解释型语言则对高级语言程序逐句解释执行。
  - 优点：程序设计的灵活性大
  - 缺点：运行效率较低
  - 常见语言：Basic、Python、Matlab

数制

数制

定义

数制也称计数制，是用一组固定的符号和统一的规则来表示数值的方法。通常采用的数制有十进制、二进制、八进制和十六进制。

在一种数制中，只能使用一组固定的数字符号来表示数目的大小。具体使用多少个数字符号来表示数目的大小，就称为该数制的基数。

数制

- 十进制 (Decimal)

  基数是 10，有 10 个数字符号，即 0，I，2，3，4，5，6，7，8，9。

- 二进制 (Binary)

  基数是 2，只有两个数字符号，即 0 和 1。

- 八进制 (Octal)

  基数是 8，它有 8 个数字符号，即 0，I，2，3，4，5，6，7。

- 十六进制 (Hexadecimal)

  基数是 16，它有 16 个数字符号，除了十进制中的 10 个数外，还使用了 6 个英文字母。16 个数字符号依次是

  $$0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ A,\ B,\ C,\ D,\ E,\ F.$$

  其中 A~F 分别代表十进制数的 10~15。

数制

二进制	十进制	八进制	十六进制
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F
10000	16	20	10

进制

在数制中，N 进制必须是逢 N 进一。

- 十进制数

$$(1010)_{10} = 1 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 0 \times 10^0$$

- 二进制数

$$(1010)_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = (10)_{10}$$

- 八进制数

$$(1010)_8 = 1 \times 8^3 + 0 \times 8^2 + 1 \times 8^1 + 0 \times 8^0 = (520)_{10}$$

- 十六进制数

$$(\mathrm{BAD})_{16} = 11 \times 16^2 + 10 \times 16^1 + 13 \times 16^0 = (2989)_{10}$$

二进制数

加法法则

$$0+0=0$$
$$0+1=1$$
$$1+0=1$$
$$1+1=10$$
$$1+1+1=10+1=11$$

例

$$\begin{array}{rr} & 1\,0\,1\,1 \\ + & 1\,0\,1\,0 \\ \hline = & 1\,0\,1\,0\,1 \end{array}$$

二进制数

减法法则

$$
\begin{aligned}
0-0&=0\\
1-0&=1\\
1-1&=0\\
0-1&=1 \quad \text{有借位，借 1 当 } (10)_2\\
0-1-1&=0 \quad \text{有借位}\\
1-1-1&=1 \quad \text{有借位}
\end{aligned}
$$

例

$$
\begin{array}{rr}
 & 1\,1\,0\,0\,0\,0\\
- & \quad 1\,0\,1\,1\,1\\
\hline
= & \quad 1\,1\,0\,0\,1
\end{array}
$$

二进制数

乘法法则

$$0\times 0=0,\quad 0\times 1=0$$
$$1\times 0=0,\quad 1\times 1=1$$

例

$$\begin{array}{rr} & 1\,1\,1\,0 \\ \times & 0\,1\,1\,0 \\ \hline & 0\,0\,0\,0 \\ & 1\,1\,1\,0 \\ & 1\,1\,1\,0 \\ + & 0\,0\,0\,0 \\ \hline & 1\,0\,1\,0\,1\,0\,0 \end{array}$$

二进制数

$$
\begin{array}{rr}
 & 1\,1\,0\,1 \\
\cline{2-2}
110\ \sqrt{} & 1\,0\,0\,1\,1\,1\,0 \\
- & \underline{\ \ 1\,1\,0} \\
 & \ \ 1\,1\,1 \\
- & \ \ \underline{1\,1\,0} \\
 & \ \ \ \ 1\,1\,0 \\
- & \ \ \ \ \underline{1\,1\,0} \\
 & \ \ \ \ \ \ 0
\end{array}
$$

图 **1:** 二进制数的除法

十进制转二进制

整数部分

除 2 取余，直至商为 0，最后将所得余数按逆序排列。

例

$$
\begin{array}{r|l|l}
2 & 23 & \\
\hline
2 & 11 & 1 \\
\hline
2 & 5 & 1 \\
\hline
2 & 2 & 1 \\
\hline
2 & 1 & 0 \\
\hline
 & 0 & 1
\end{array}
$$

图 2: $(23)_{10} = (10111)_2$

十进制转二进制

小数部分

乘 2 取整数，若小数部分是 5 的倍数，则以最后小数部分为 0 为止，否则以约定的精确度为准，最后将所取整数按顺序排列。

例

$$
\begin{array}{rl}
 & 0.25 \\
\times & \quad 2 \\
\hline
 & 0.50 \quad \text{取整数位0} \\
\times & \quad 2 \\
\hline
 & 1.00 \quad \text{取整数位1}
\end{array}
$$

图 3: $(0.25)_{10} = (0.01)_2$

十进制转二进制

例

将十进制数 125.24 转换为二进制数 (取四位小数)。

十进制转二进制

例

将十进制数 125.24 转换为二进制数 (取四位小数)。

2	125	
2	62	1
2	31	0
2	15	1
2	7	1
2	3	1
2	1	1
	0	1

0.24	
× 2	
0.48	取整数位0
× 2	
0.96	取整数位0
× 2	
1.92	取整数位1
× 2	
1.84	取整数位1

十进制转二进制

例

将十进制数 125.24 转换为二进制数 (取四位小数)。

$$
\begin{array}{r|l|l}
2 & 125 & \\
2 & 62 & 1 \\
2 & 31 & 0 \\
2 & 15 & 1 \\
2 & 7 & 1 \\
2 & 3 & 1 \\
2 & 1 & 1 \\
 & 0 & 1
\end{array}
$$

$$
\begin{array}{rl}
 & 0.24 \\
\times & \quad 2 \\
\hline
 & 0.48 \quad \text{取整数位0} \\
\times & \quad 2 \\
\hline
 & 0.96 \quad \text{取整数位0} \\
\times & \quad 2 \\
\hline
 & 1.92 \quad \text{取整数位1} \\
\times & \quad 2 \\
\hline
 & 1.84 \quad \text{取整数位1}
\end{array}
$$

结果为

$$(125.24)_{10} = (1111101.0011)_2.$$

二进制转十进制

例

将 $(1101.101)_2$ 转换为十进制数。

二进制转十进制

例

将 $(1101.101)_2$ 转换为十进制数。

$$
\begin{aligned}
& (1101.101)_2 \\
= \quad & 1\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3} \\
= \quad & 8+4+1+0.5+0.125 \\
= \quad & 13.625
\end{aligned}
$$

二进制转十六进制

例

将 $(11\ 1010\ 1111\ 0001\ 0111)_2$ 转换为十六进制数。

二进制转十六进制

例

将 $(11\ 1010\ 1111\ 0001\ 0111)_2$ 转换为十六进制数。

二进制数	11	1010	1111	0001	0111
十六进制数	3	A	F	1	7

结果为

$$(11\ 1010\ 1111\ 0001\ 0111)_2 = (3AF17)_{16}$$

汇编语言简介

机器语言

定义

机器语言是机器指令的集合。机器指令是计算机可以正确执行的命令，它是一组二进制数字。计算机将其转变为一组高低电平，以使电子器件收到驱动，进行运算。

机器语言

定义

机器语言是机器指令的集合。机器指令是计算机可以正确执行的命令，它是一组二进制数字。计算机将其转变为一组高低电平，以使电子器件收到驱动，进行运算。

例

8086CPU 完成运算 $s = 768 + 12288 - 1280$ 的机器码如下：

1011 0000 0000 0000 0000 0011
0000 0101 0000 0000 0011 0000
0010 1101 0000 0000 0000 0101

机器语言

定义

机器语言是机器指令的集合。机器指令是计算机可以正确执行的命令，它是一组二进制数字。计算机将其转变为一组高低电平，以使电子器件收到驱动，进行运算。

例

8086CPU 完成运算 $s = 768 + 12288 - 1280$ 的机器码如下：

1011 0000 0000 0000 0000 0011
0000 0101 0000 0000 0011 0000
0010 1101 0000 0000 0000 0101

若将程序错写成以下形式，请指出错误：

1011 0000 0000 0000 0000 0011
0000 0101 0000 0000 0011 0000
0001 0110 1000 0000 0000 0101

汇编语言简介

定义：汇编语言

汇编语言的主体是汇编指令。汇编指令和机器指令的差别在于指令的表示方法上。汇编指令是机器指令便于记忆的书写形式。

汇编语言简介

定义：汇编语言

汇编语言的主体是汇编指令。汇编指令和机器指令的差别在于指令的表示方法上。汇编指令是机器指令便于记忆的书写形式。

例

操作：寄存器 BX 的内容送到 AX 中

机器指令：1000 1001 1101 1000

汇编指令：mov ax, bx

汇编语言简介

问题

计算机只能读懂机器指令，那么如何让计算机执行汇编指令编写的程序呢?

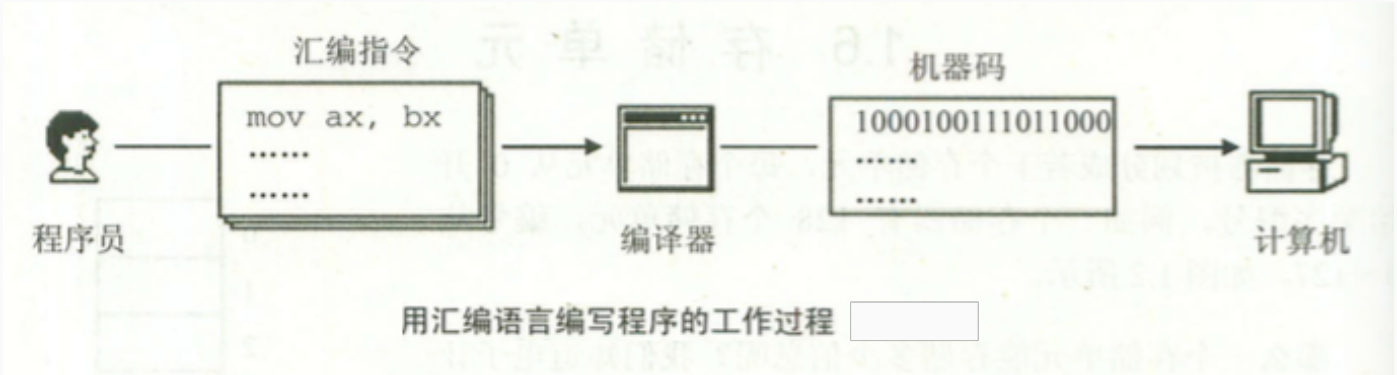
汇编语言简介

问题

计算机只能读懂机器指令，那么如何让计算机执行汇编指令编写的程序呢?

需要用到一个能将汇编指令转换为机器指令的翻译程序，即编译器。

程序员用汇编语言写出源程序，再用编译器将其编译为机器码，由计算机最终执行。



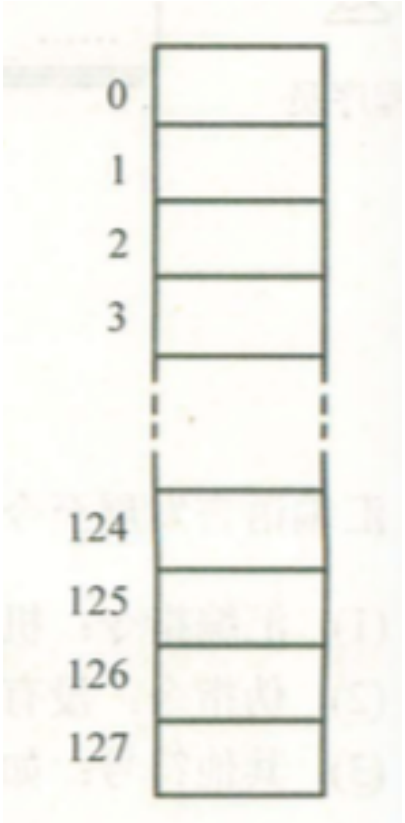
用汇编语言编写程序的工作过程

存储单元

存储器被划分为若干个存储单元，每个存储单元从 0 开始顺序编号。

例

假设有一个存储器，编号从 0～127，如下图：



存储单元

定义

- 计算机的最小信息单位是 Bit，也就是一个二进制位。
- 8 个 Bit 组成 1 个 Byte，即一个字节。一个存储单元可存储一个字节。

  若一个存储器有 128 个存储单元，它可以存储 128 个字节。
- 存储器的容量以字节为最小单位来计算。

  对于拥有 128 个存储单元的存储器，其容量为 128 个字节。

对于大容量的存储器，还用以下单位来计算容量（以下用 B 来代表 Byte）：

$$1KB = 2^{10}B = 1024B, \quad 1MB = 1025KB,$$
$$1GB = 1024KB, \quad 1TB = 1025GB,$$

CPU 对存储器的读写

- CPU 要从内存中读数据，首先要指定单元地址。也就是说要先确定要读哪个单元中的内容。
- 存储器不止一种。CPU 在读写数据时还要指明，对哪一个存储器进行操作，进行哪种操作，是从中读取数据，还是向里面写入数据。

CPU 对存储器的读写

CPU 要想进行数据的读写，必须与外部器件（芯片）进行以下 3 类信息的交互：

- 存储单元的地址（地址信息）；
- 器件的选择，读或写的命令（控制信息）；
- 读或写的数据（数据信息）。

CPU 对存储器的读写

问题

CPU 通过什么将地址、数据和控制信息传给存储器芯片呢?

CPU 对存储器的读写

问题

CPU 通过什么将地址、数据和控制信息传给存储器芯片呢?

计算机传输的信息都是电信号，电信号当然要用导线传送。

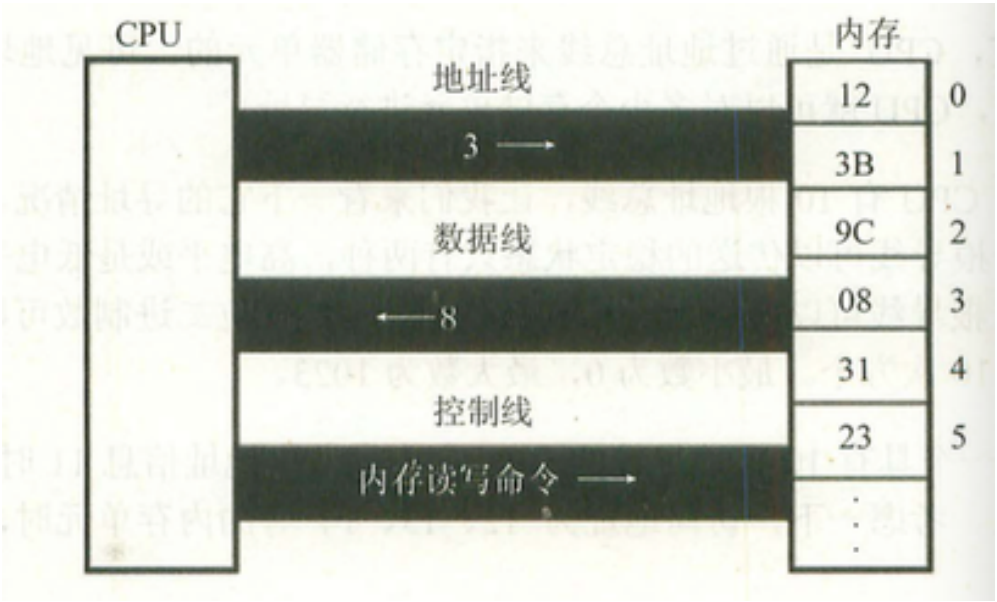
定义：总线

在计算机中专门有连接 CPU 和其它芯片的导线，通常称为总线。根据传送信息的不同，从逻辑上分为三类:

- 地址总线
- 控制总线
- 数据总线

CPU 从内存中读取数据

例



1. CPU 通过地址总线将地址信息 3 发出；
2. CPU 通过控制总线发出内存读命令，选中存储器芯片，并通知它，将要从中读取数据；
3. 存储器将 3 号单元的数据 8 通过数据总线送入 CPU。

CPU 对存储器的读写

写操作与读操作的步骤类似。如向 3 号单元写入数据 26。

1. CPU 通过地址总线将地址信息 3 发出；
2. CPU 通过控制总线发出内存写命令，选中存储器芯片，并通知它，将要从中写入数据；
3. CPU 通过数据总线将数据 26 送入内存的 3 号单元中。

CPU 对存储器的读写

问题

我们知道了 CPU 如何进行数据的读写。那么，我们又如何命令计算机进行数据的读写呢?

CPU 对存储器的读写

问题

我们知道了 CPU 如何进行数据的读写。那么，我们又如何命令计算机进行数据的读写呢?

要让计算机工作，应向它输入能驱动其进行工作的电平信息，即机器码。

机器指令：1010 0001 0000 0011 0000 0000

汇编指令：mov ax, [3]

含义：传送 3 号单元的内容入 ax。

地址总线

- CPU 通过地址总线指定存储器单元。由此可见，地址总线能传送多少个不同的信息，CPU 就可以对多少个存储单元进行寻址。

地址总线

- CPU 通过地址总线指定存储器单元。由此可见，地址总线能传送多少个不同的信息，CPU 就可以对多少个存储单元进行寻址。
- 现假设一个 CPU 有 10 根地址总线，可以传送 10 位二进制数据，共 2^{10} 个不同数据，最小为 0，最大为 1023。

地址总线

- CPU 通过地址总线指定存储器单元。由此可见，地址总线能传送多少个不同的信息，CPU 就可以对多少个存储单元进行寻址。
- 现假设一个 CPU 有 10 根地址总线，可以传送 10 位二进制数据，共 2^{10} 个不同数据，最小为 0，最大为 1023。
- 一个 CPU 有 N 根地址线，则称 CPU 的地址总线的宽度为 N，最多可以寻找 2^N 个内存单元。

地址总线

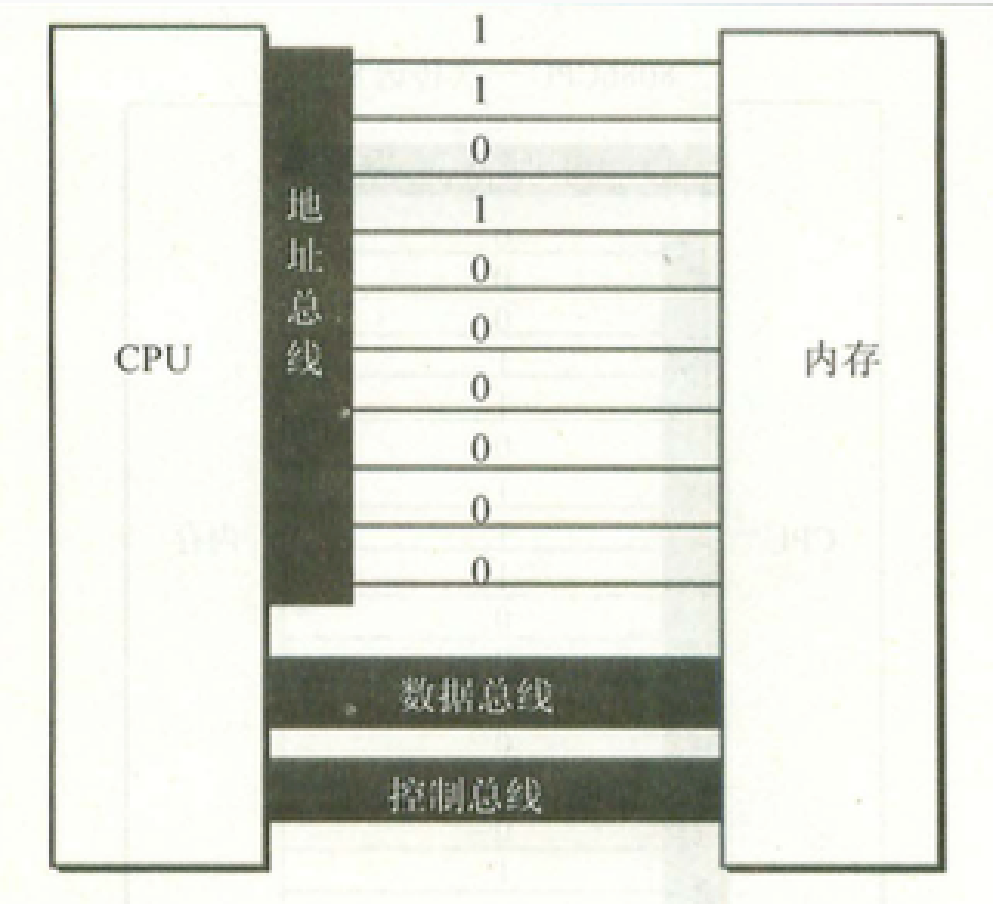


图 4: 地址总线发送的地址信息

数据总线

CPU 与内存和其它器件之间的数据传输通过数据总线来进行。数据总线的宽度决定了 CPU 与外界的数据传输速度。

数据总线

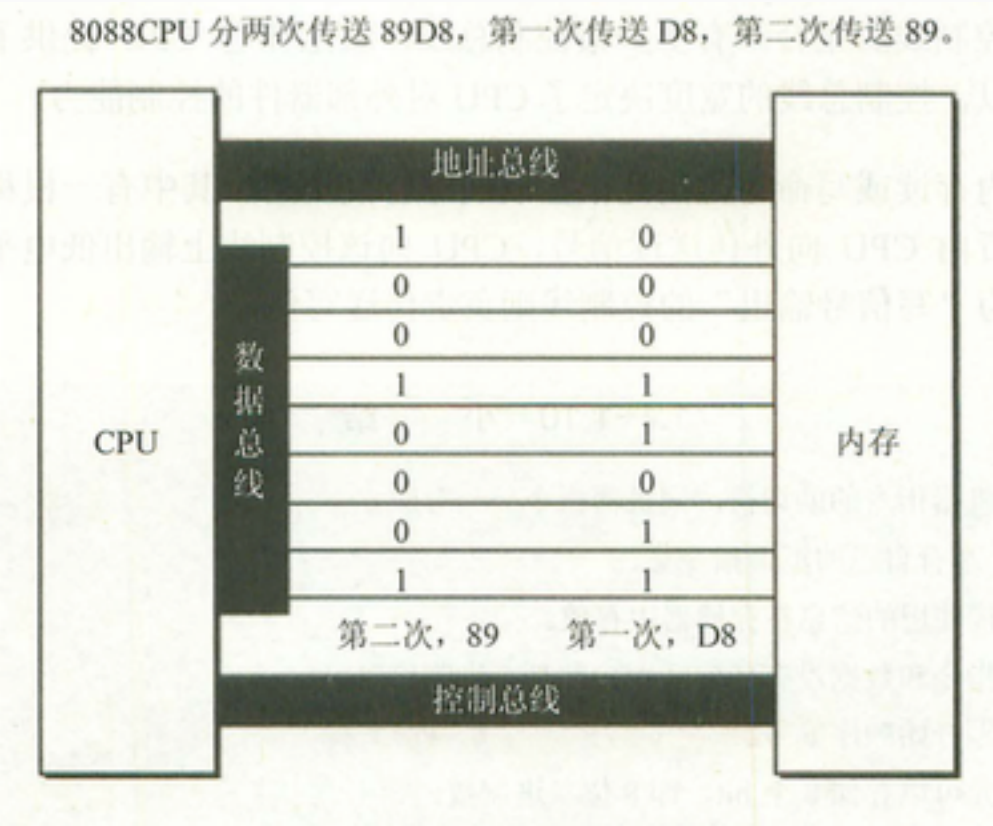


图 5: 8 根数据总线一次可传输一个字节

数据总线

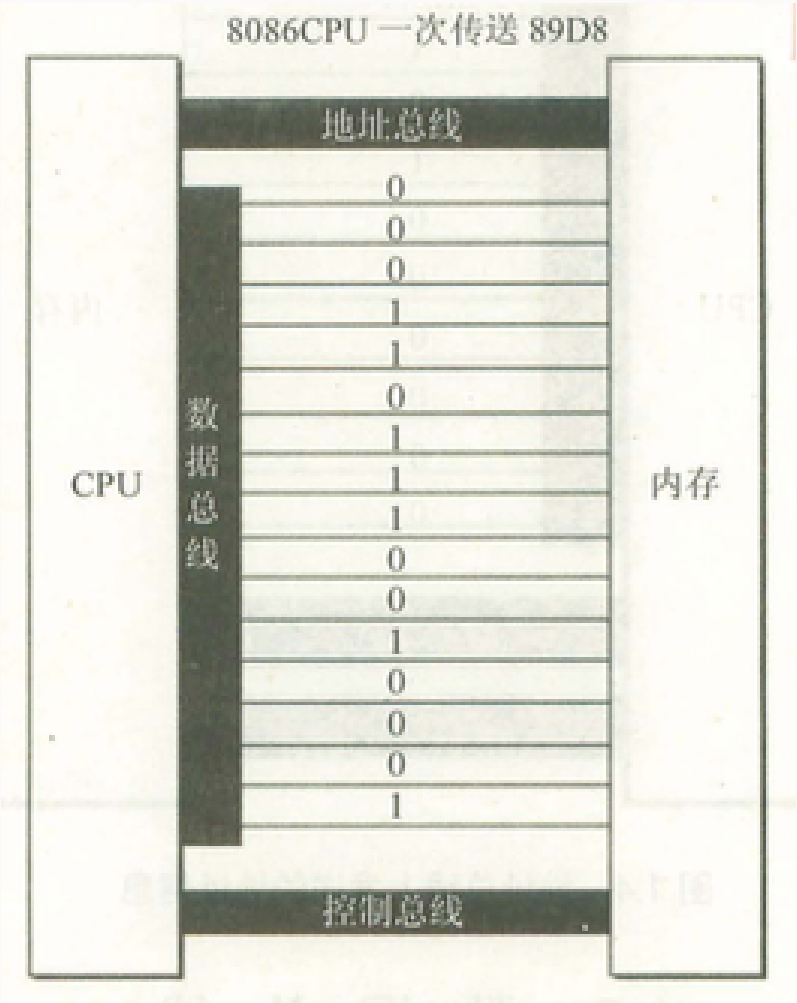


图 **6:** 16 根数据总线一次可传输两个字节

控制总线

CPU 对外部器件的控制通过控制总线来进行。有多少根控制总线，就意味着 CPU 提供了对外部器件的多少种控制。因此，控制总线的宽度决定了 CPU 对外部器件的控制能力。

寄存器

一个典型的 CPU 由运算器、控制器、寄存器等器件构成，这些器件通过内部总线相连。内部总线实现 CPU 内部各个器件之间的联系，而外部总线实现 CPU 与主板上其它器件的联系。

在 CPU 中：

- 运算器进行信息处理；
- 寄存器进行信息存储；
- 控制器控制各种器件进行工作；
- 内部总线连接各种器件，在它们之间进行数据的传送。

寄存器

寄存器是 CPU 中程序员可以用指令读写的部件，程序员通过改变各种寄存器中的内容来实现对 CPU 的控制。

不同的 CPU，寄存器的个数、结构不尽相同。8086CPU 有 14 个寄存器，每个寄存器都有一个名称，分别是

AX、BX、CX、DX、SI、DI、SP、BP、IP、CS、SS、DS、ES、PSW。

通用寄存器

8086CPU 的所有寄存器都是 16 位的，可以存放两个字节。

AX、BX、CX、DX 这四个寄存器通常用来存放一般性的数据，被称为通用寄存器。

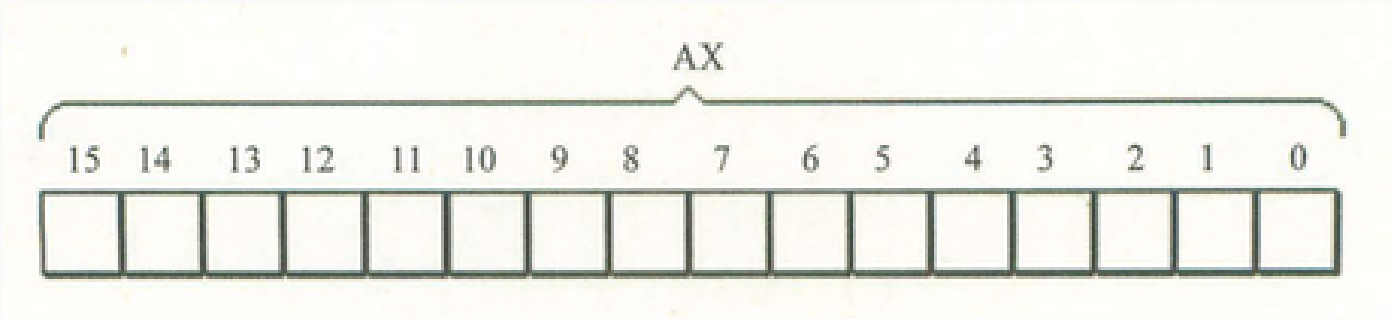


图 7: 16 位寄存器的逻辑结构

通用寄存器

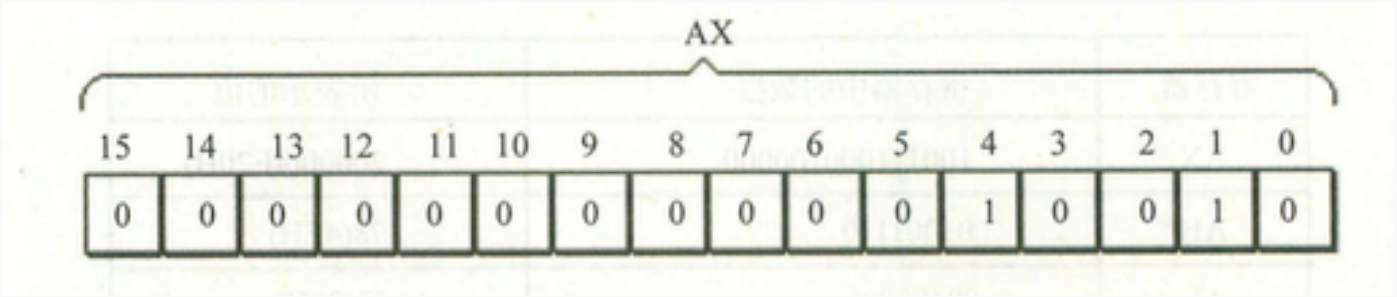


图 **8:** $(10010)_2$ 在寄存器 AX 中的存储

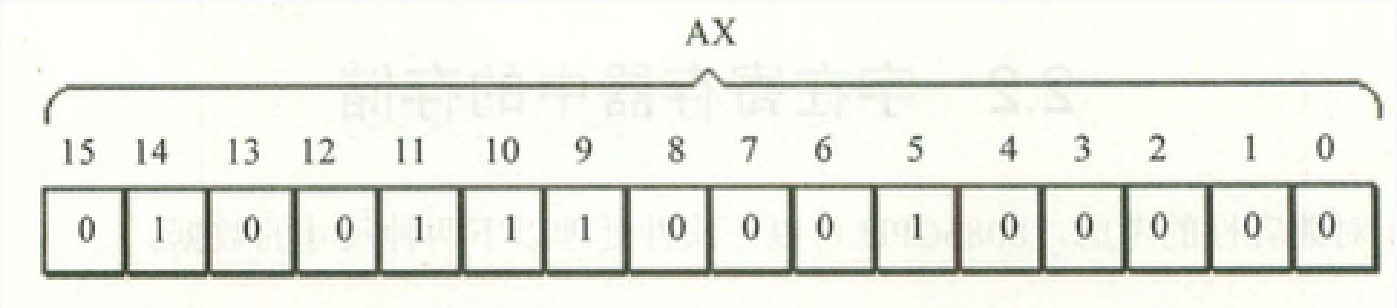


图 **9:** $(100111000100000)_2$ 在寄存器 AX 中的存储

通用寄存器

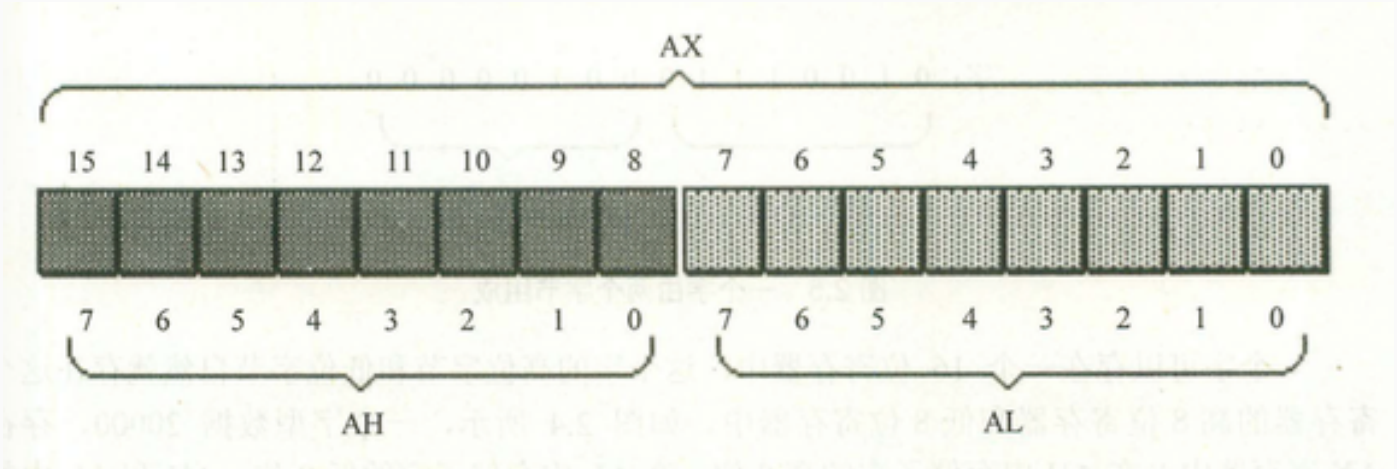


图 **10:** 16 位寄存器可分为两个 8 位寄存器

几条汇编指令

汇编指令	控制 CPU 完成的操作	用高级语言的语法描述
mov ax,18	将 18 送入寄存器 AX	AX=18
mov ah,78	将 78 送入寄存器 AH	AH=78
add ax,8	将寄存器 AX 中的数值加上 8	AX=AX+8
mov ax,bx	将寄存器 BX 中的数据送入寄存器 AX	AX=BX
add ax,bx	将 AX 和 BX 中的数值相加，结果存在 AX 中	AX=AX+BX

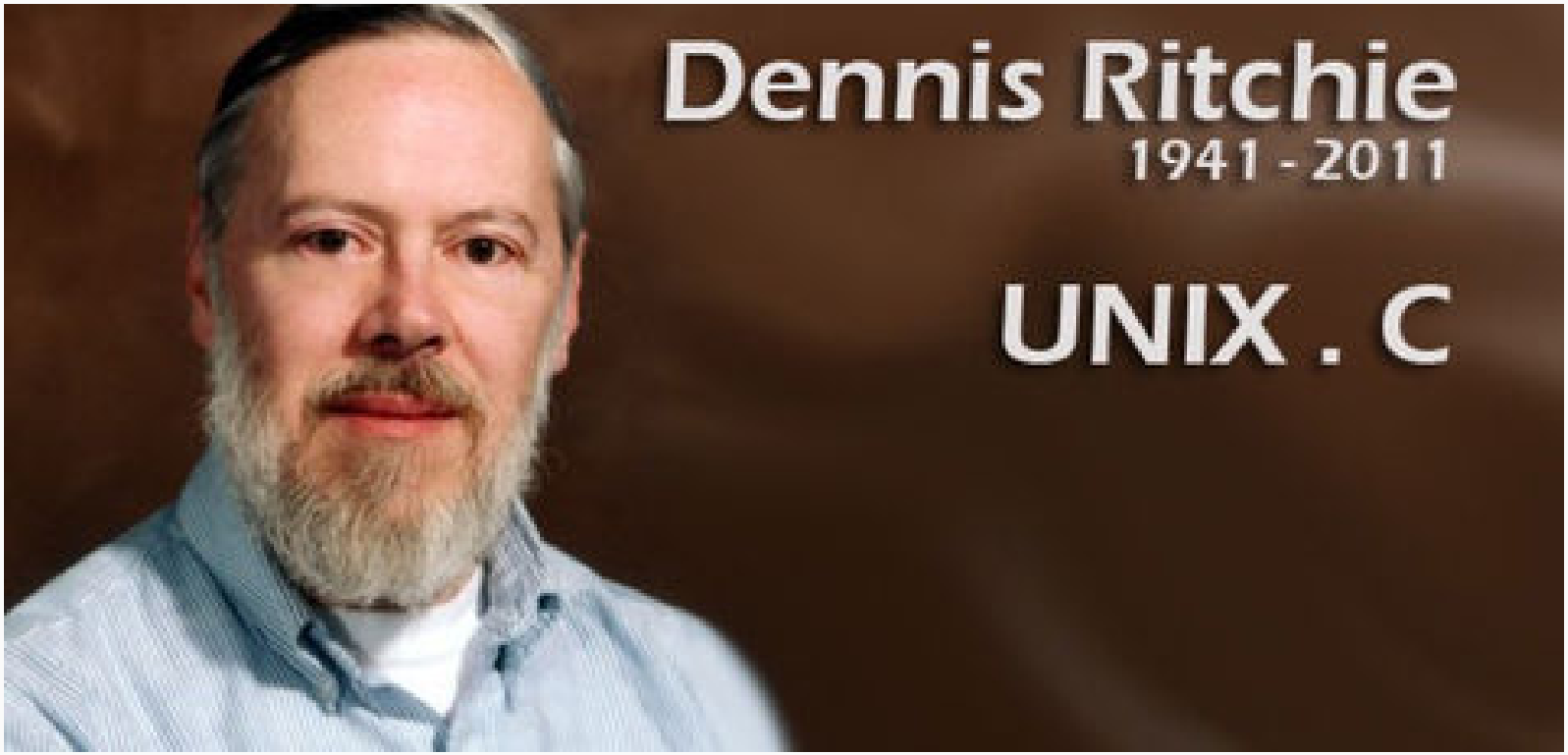
图 11: 汇编指令举例

C 语言简介

C 的起源

- 产生时间：1972-1973 年
- 产生地点：美国贝尔实验室
- 创始人：Dennis Ritchie & Ken Thompson
- 目的：改写 Unix 系统
- 荣誉：美国国家技术奖章 (1999)

C 的起源



C 的起源

图 **12:** Ken Thompson (1942-)

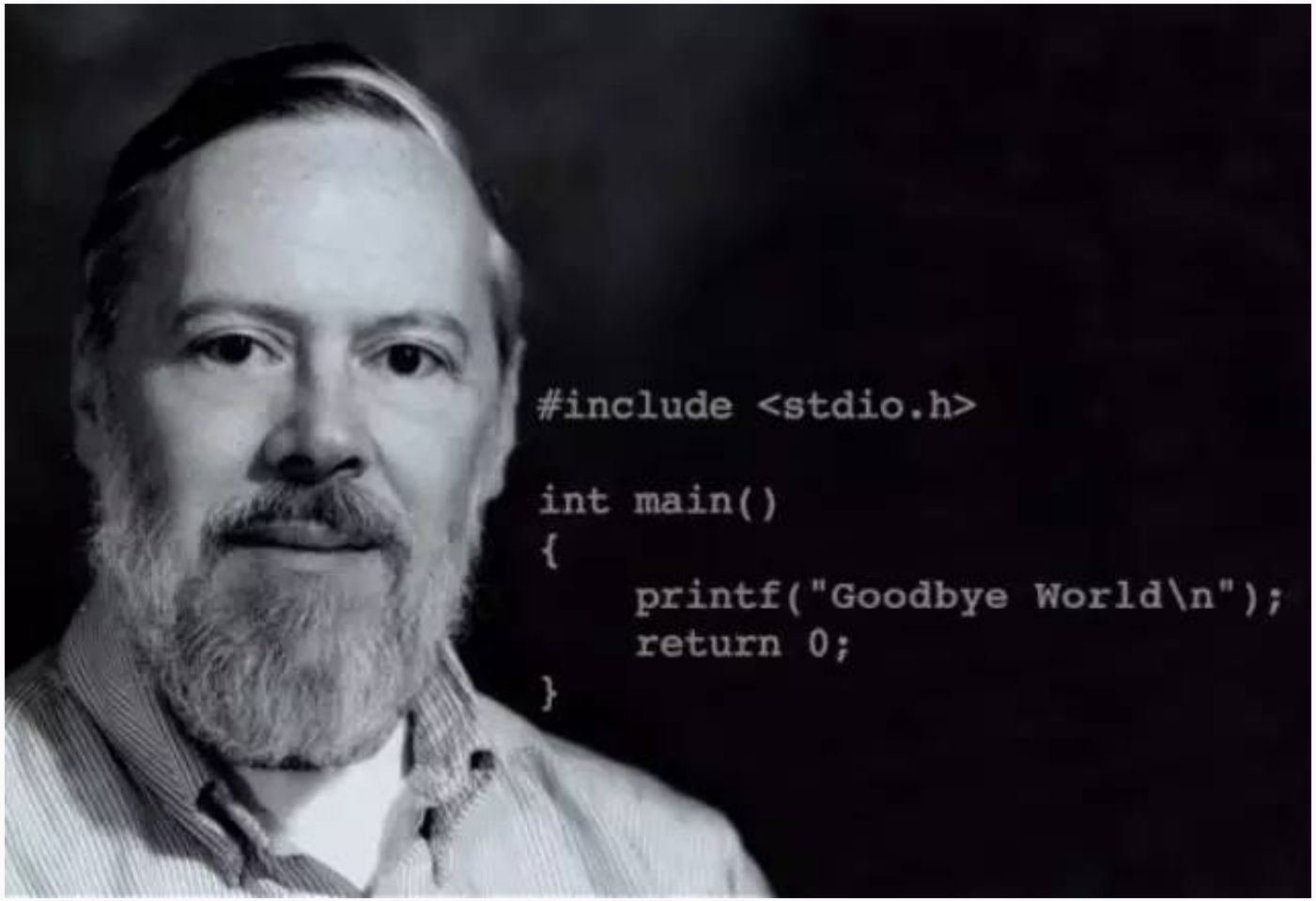
C 的起源

图 **13:** Dennis Ritchie 和 Ken Thompson(1972 年)

C 的起源

1983 年，Dennis Ritchie 和 Ken Thompson 一起获得了图灵奖，理由是：“研究发展了通用的操作系统理论，尤其是实现了 Unix 操作系统”。

关于 Ritche



关于 Ritche

“当乔布斯去世时，享受到了声势浩大的追思。相形之下，里奇先生对当代科技进程做出了更大的贡献，可公众甚至不知道他是谁，这十分不公平。”

“如果说，乔布斯是可视化产品中的国王，那么里奇就是不可见王国中的君主。乔布斯的贡献在于，他如此了解用户的需求和渴求，以至于创造出了让当代人乐不思蜀的科技产品。然而，却是里奇先生为这些产品提供了最核心的部件，人们看不到这些部件，却每天都在使用着。”

“牛顿说他是站在巨人的肩膀上，如今，我们都站在里奇的肩膀上。”

C 的地位

- C 拥有汇编语言的力量和便利性，其运行方式更接近于硬件系统；
- C 所提供的数据结构，力发千钧，足以贯穿所有高层和底层的语言；
- C 的开发是科技史上不可磨灭的伟大贡献，因为这个语言把握住了计算机科技中一个至关重要的并且是恰到好处的中间点，一方面它具备搭建高层产品的能力，另一方面又能够对于底层数据进行有效控制。正是由于这种关联性和枢纽性作用，决定了 C 所导向的近三十年来计算机编程主流方式。

C 的优点

C 语言的优点

- 设计特性：融合了控制特性，使得用户可以结构化编程及模块化设计，程序更可靠、更易懂。
- 高效性：程序紧凑、运行速度快，可表现出只有汇编语言才具有的精细控制能力。
- 可移植性：在一个系统上编写的 C 程序经过很少改动或不经修改便可在其他系统上运行。
- 强大的功能与灵活性
  - Unix 操作系统是用 C 编写的。
  - 很多语言 (如 Fortran、Python、Pascal 等) 的编译器或解释器是用 C 编写的。
- 面向程序员
  - 允许访问硬件，并操纵内存中的特定位。
  - 具有丰富的运算符供选择，让程序员能简洁地表达自己的意图。

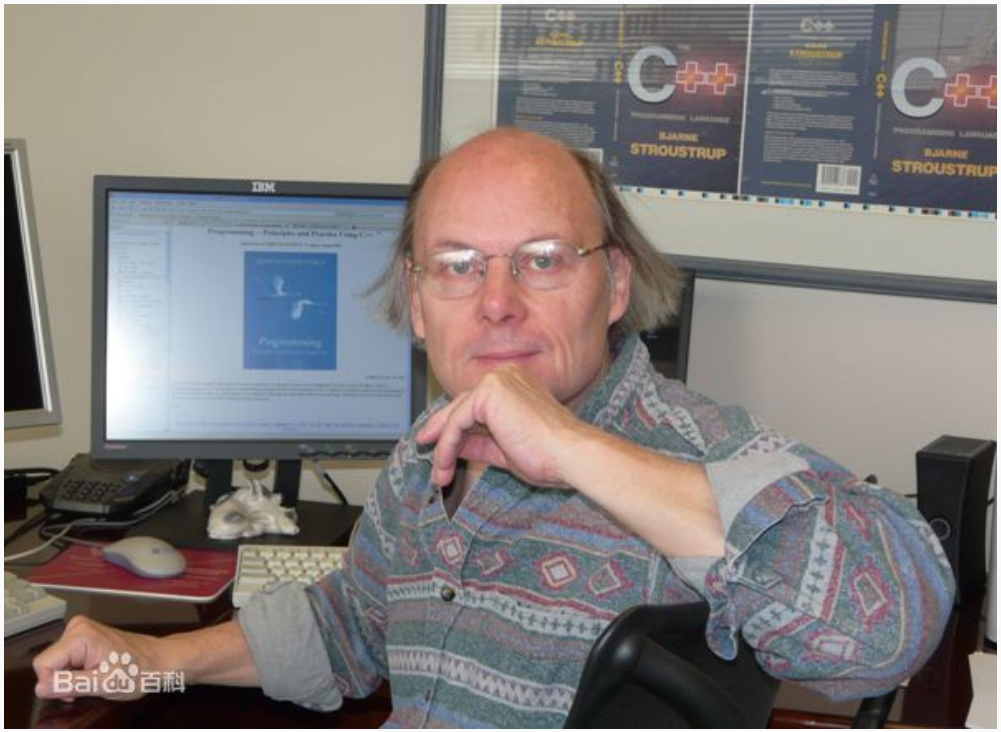
C 的缺点

- 在表达方面的自由会增加风险。
- 对指针的使用，可能导致你会犯难以追踪的编程错误。
  自由的代价是永远的警惕。
- 简洁性与丰富的运算符相结合，可能会产生极难理解的代码。
  含糊代码竞赛 `www.ioccc.org`

C++ 简介

C++ 起源

- 产生时间： 20 世纪 80 年代
- 产生地点：美国贝尔实验室
- 创始人： Bjarne Stroustrup



C++ 发展

C++ 从最初的 C with class，经历了从 C++98、C++03、C++11、C++14 再到 C++17 多次标准化改造，功能得到了极大的丰富，已经演变为一门集面向过程、面向对象、函数式、泛型编程等多种编程范式的复杂编程语言。

C++ 特点

C++ 融合了 3 种不同的编程方式：

- C 语言代表的过程性语言
- C++ 在 C 语言基础上添加的类代表的面向对象语言 (继承、封装、多态)
- C++ 模板支持的泛型编程。

C++ 的应用领域

- 系统层软件开发
- 服务器程序开发
- 游戏、网络、分布式、云计算
- 科学计算

C++ 对 C 的增强

- C 是一个结构化语言，首要考虑的是如何通过一个过程，对输入（或环境条件）进行运算处理得到输出。
- C++ 首要考虑的是如何构造一个对象模型，让构造的模型能够契合与之对应的问题域，通过获取对象的状态信息得到输出或实现过程（事物）控制。

因此，C 和 C++ 的最大区别在于解决问题的思想不一样：

- C 面向过程
- C++ 面向对象

C++ 对 C 的增强

C++ 对 C 的增强表现在六个方面：

1. 类型检查更为严格
2. 增加了面向对象的机制
3. 增加了泛型编程的机制 (Template)
4. 增加了异常处理
5. 增加了运算符重载
6. 增加了标准模板库 (STL)

编译 C 程序的工作原理

编译 C 程序的工作原理

C 是一种高级语言，它需要编译器将其转换为可执行代码，以使得程序能在机器上运行。

以下介绍在 MAC 或 Linux 上使用 gcc 编译器的几个步骤。

编译 C 程序的工作原理

(1) 首先使用编辑器（如 vi 或 emacs 等）创建一个 C 程序，并将其保存为 hello.c。

```
$ emacs hello.c
```

编译 C 程序的工作原理

在编辑界面输入以下内容:

```
/*
 *   A simple c code
 */
#include <stdio.h>
int main(void)
{
   printf("Hello World!\n");
   return 0;
}
```

编译 C 程序的工作原理

(2) 然后用以下命令编译，并查看当前目录下的文件。

```
$ gcc -Wall hello.c -o hello
$ ls
  hello     hello.c
```

- 选项 `-Wall` 启动所有编译器的警告信息。建议使用该选项以生成更好的代码。
- 选项 `-o` 用来制定输出文件名。如果缺省该选项，则输出文件将默认为 `a.out`。

编译 C 程序的工作原理

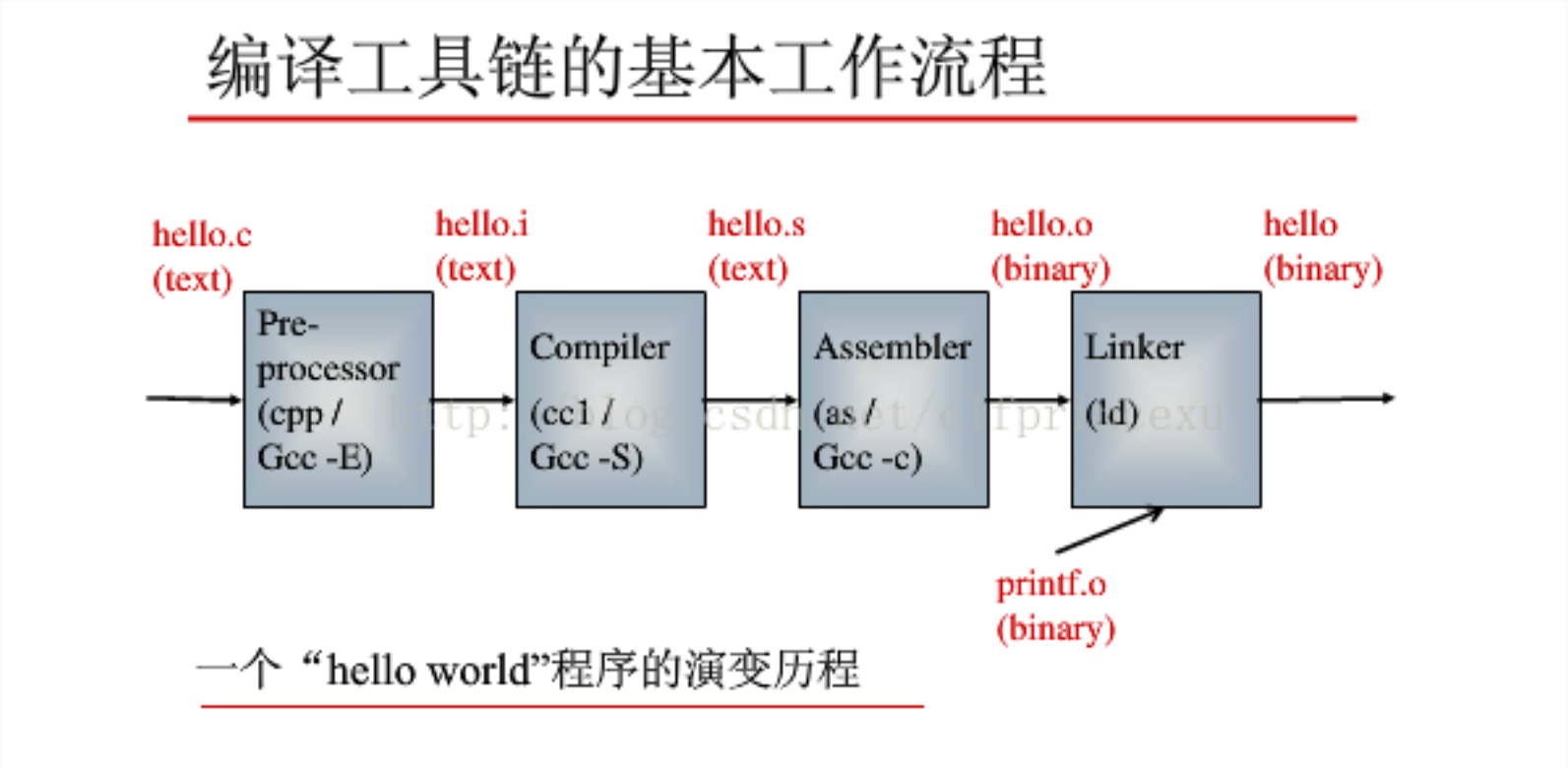
(3) 编译通过后，将会生成可执行文件。可用以下命令来运行：

```
$ ./hello
Hello World!
```

编译 C 程序的工作原理

编译器将一个 C 程序转换为一个可执行文件，需经历了 4 个阶段：

- 预处理
- 编译
- 汇编
- 链接



编译 C 程序的工作原理

执行以下命令，会在当前目录下生成所有的中间文件以及可执行文件。

```
$ gcc -Wall -save-temps hello.c -o hello
$ ls
hello      hello.c    hello.o
hello.i    hello.s
```

编译 C 程序的工作原理

(1) 预处理阶段

```
$ gcc -E hello.c -o hello.i
```

在该阶段，

- 去掉注释
- 宏的展开
- 头文件的展开
- 选项 `-E` 的作用是让 gcc 在预处理结束后停止编译过程。

编译 C 程序的工作原理

```
$ less hello.i
$ ls
...
extern int printf (const char *__restrict __format, ...)
;
...
# 868 "/usr/include/stdio.h" 3 4
# 5 "hello.c" 2
# 5 "hello.c"

int main(void)
{
  printf("Hello World!\n");
  return 0;
}
```

编译 C 程序的工作原理

(2) 编译阶段

```
$ gcc -S hello.i -o hello.s
```

在该阶段，

- 检查代码的规范性、是否有语法错误等
- 检查无误后，将代码翻译成汇编语言
- 选项 `-S` 只进行编译，不进行汇编，生成汇编代码。

编译 C 程序的工作原理

```
$ less hello.s
...
movl    $0, -4(%rbp)
movl    $5, -8(%rbp)
movl    $4, -12(%rbp)
movl    -8(%rbp), %eax
addl    -12(%rbp), %eax
movl    %eax, %esi
movb    $0, %al
callq   _printf
xorl    %esi, %esi
...
```

(3) 汇编阶段

```
$ gcc -c hello.s -o hello.o
```

在该阶段，

- 将通过汇编器将 hello.s 转换成二进制机器指令文件hello.o。
- 只会将现有代码转换成机器语言，而诸如 printf() 的函数调用则不会。
- 选项 -c 的作用是将汇编代码转换为二进制目标代码。

编译 C 程序的工作原理

```
$ less hello.o
<CF><FA><ED><FE>^G^@^@^A^C^@^@^@^A^@^@^@^D^@
^@^@^@^B^@^@^@ ^@^@^@^@^@^@^Y^@^@^@<88>^A^@^
@^@^@^@^@^@^
...
```

编译 C 程序的工作原理

(4) 链接阶段

这里涉及一个重要的概念：函数库。

(4) 链接阶段

这里涉及一个重要的概念：函数库。

问题

在 hello.c中，并没有定义 printf() 的函数实现，并且在 stdio.h 也只有函数的声明，没有定义函数的实现。那么，是在哪里实现了 printf() 呢?

(4) 链接阶段

这里涉及一个重要的概念：函数库。

问题

在 hello.c中，并没有定义 printf() 的函数实现，并且在 stdio.h 也只有函数的声明，没有定义函数的实现。那么，是在哪里实现了 printf() 呢?

事实上，系统将这些函数的定义编译后放在了相应的函数库中。在没有特别指定时，gcc 会到系统默认的搜索路径 /usr/lib 下进行查找，连接到相关的库函数后，就能使用 printf()了，而这就是链接的作用。

函数库

函数库一般分为静态库和动态库。

定义：静态库

在链接阶段，将汇编生成的目标文件.o 与引用到的库文件一起链接打包到可执行文件，对应的链接方式成为静态链接。

定义：动态库

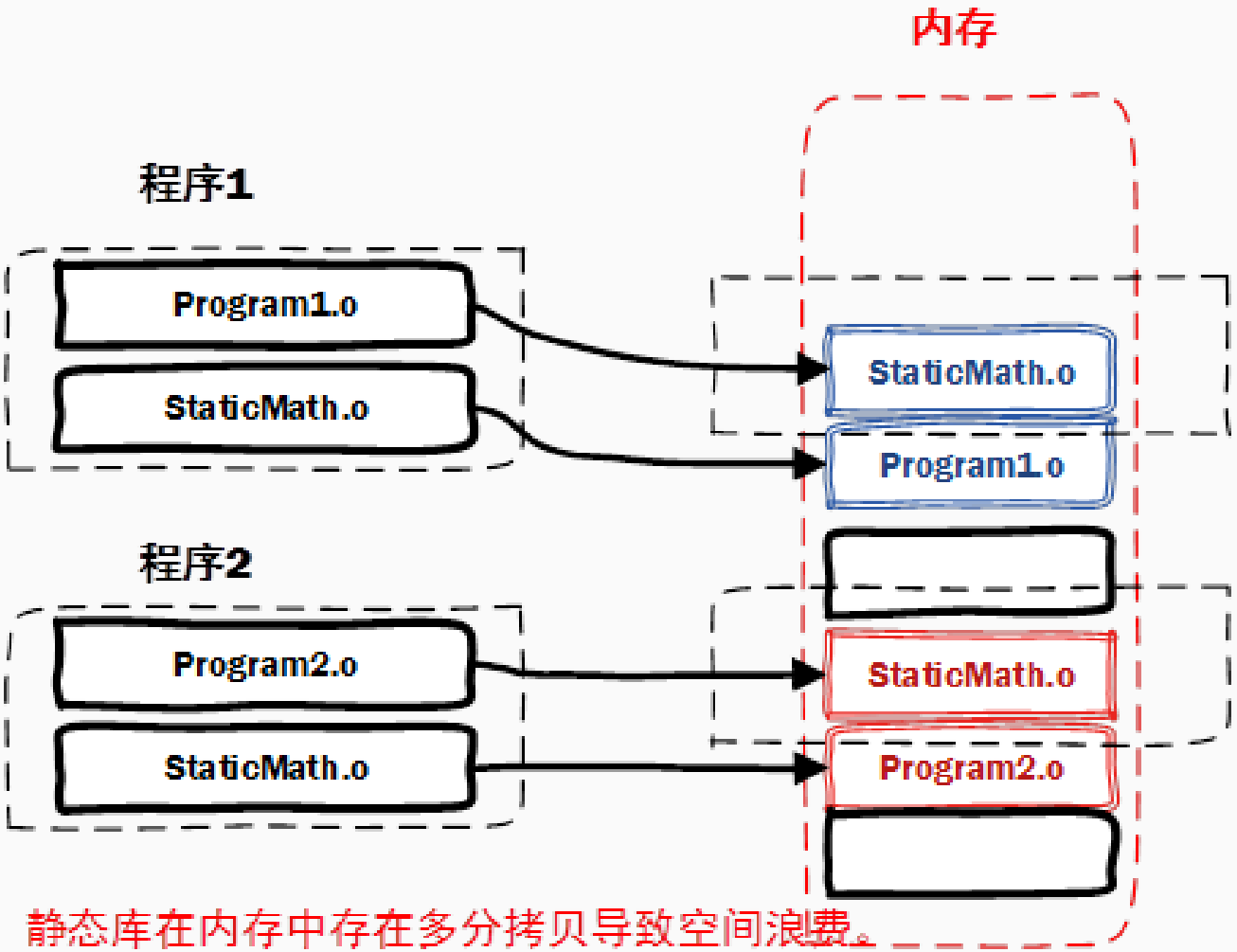
动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入。

静态库

静态库的特点

- 一个静态库可以简单看成是一组目标文件（.o/.obj 文件）的集合，即很多目标文件经过压缩打包后形成的一个文件。
- 静态库对函数库的链接是放在编译时期完成的
- 程序在运行时与函数库再无瓜葛，移植方便
- 浪费空间与资源，因为所有相关的目标文件与牵涉到的函数库被链接合并成一个可执行文件
- 后缀名为 `.a` (linux) 或 `.lib` (windows)

静态库



静态库在内存中存在多分拷贝导致空间浪费。

假如，静态库占用1M内存，有2000个这样的程序，将占用近2GB的空间~~~~~

静态库

静态库的缺点

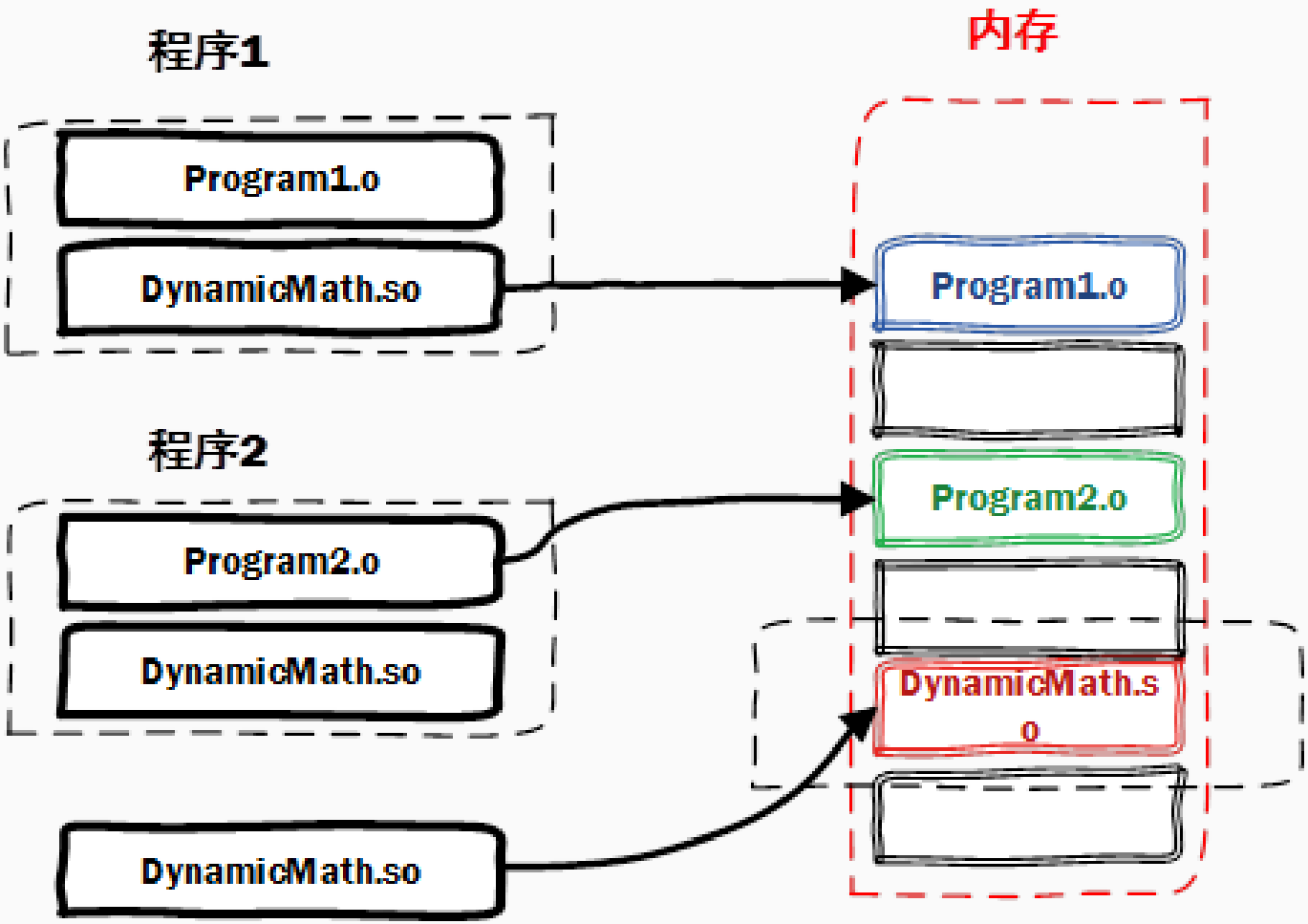
- 空间浪费
- 静态库对程序的更新、部署和发布页会带来麻烦。如果静态库 liba.lib 更新了，所以使用它的应用程序都需要重新编译、发布给用户（对于玩家来说，可能是一个很小的改动，却导致整个程序重新下载，全量更新）。

动态库

动态库的特点

- 不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例，规避了空间浪费问题。
- 动态库在程序运行是才被载入，也解决了静态库对程序的更新、部署和发布页会带来麻烦。用户只需要更新动态库即可，增量更新。
- 后缀名为 `.so` (linux) 或 `.dll` (windows)
- gcc 在编译时默认使用动态库

动态库



动态库在内存中只存在一份拷贝，避免了静态库浪费空间的问题。

函数库的创建与使用

设有 4 个文件：

- `hello.h`
- `hello1.c`
- `hello2.c`
- `main.c`

我们来看看如何创建静态库与动态库，并如何使用它们。

函数库的创建与使用

hello.h

```
#ifndef _HELLO_H_
#define _HELLO_H_

#include <stdio.h>

void hello1();
void hello2();

#endif
```

函数库的创建与使用

hello1.c

```c
#include "hello.h"

void hello1()
{
  printf("Hello World!\n");
}
```

hello2.c

```c
#include "hello.h"

void hello2()
{
  printf("Hello Guys!\n");
}
```

函数库的创建与使用

main.c

```c
#include "hello.h"

int main(void)
{
  hello1();
  hello2();

  return 0;
}
```

静态库的创建与使用

```
# 将 hello.c 编译成 hello.o（静态库和动态库都由 .o 文件生成）
$ gcc -c hello1.c hello2.c

# 为遵循 linux 中静态库的命名规范，静态库命名为 libhello.a
$ ar -crv libhello.a hello1.o hello2.o

# 将 main.c 与静态库连接，生成可执行文件 main
$ gcc main.c -L. -lhello -o main

# 运行 main
$ ./main
```

动态库的创建与使用

```
# 将 hello.c 编译成一个动态库 libhello.so
$ gcc hello1.c hello2.c -fPIC -shared -o
libhello.so

# 将 main.c 与动态库连接，生成可执行文件 main
$ gcc main.c -L. -lhello -o main

# 运行 main
$ ./main
```