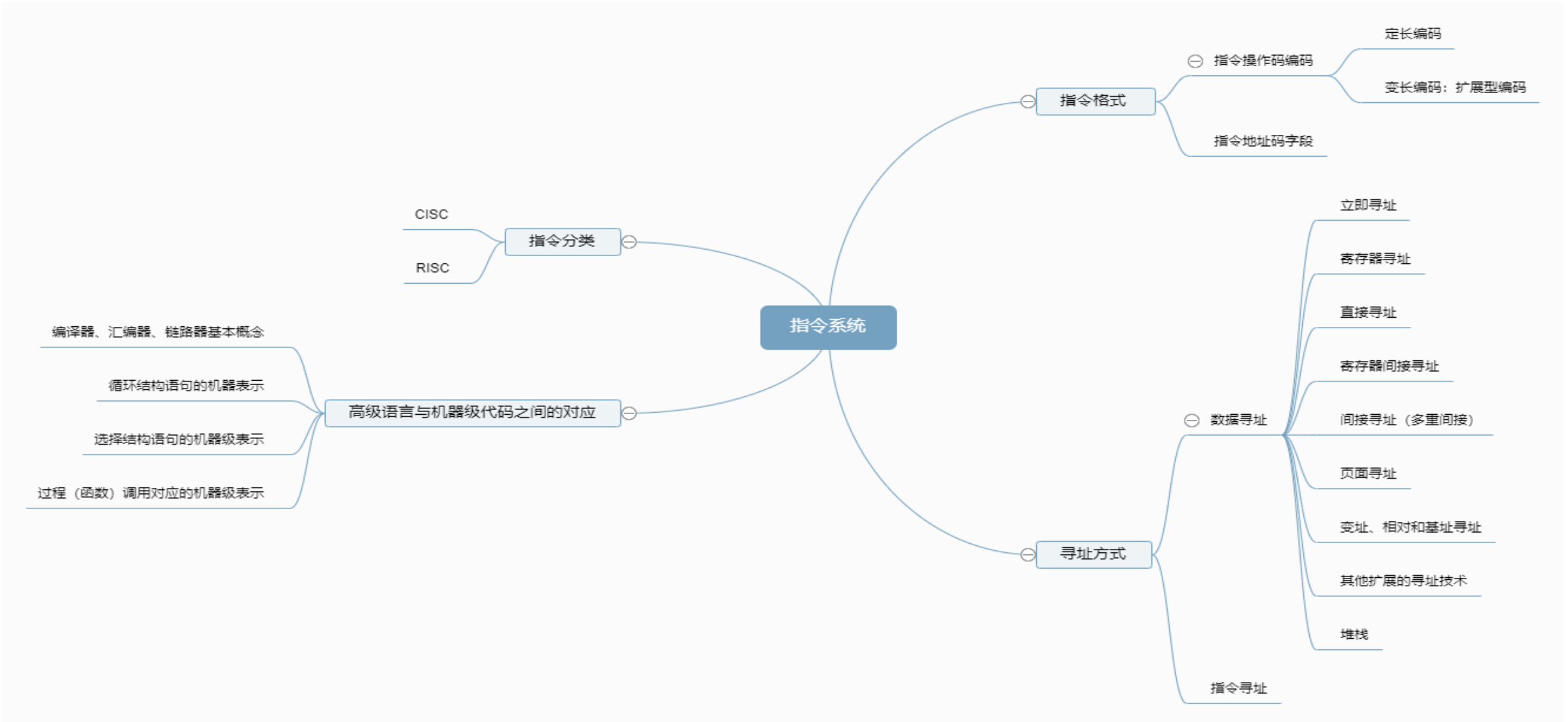
# 计算机组成与结构-指令系统

人工智能专业

主讲教师：王　娟

# 指令系统思维导图


指令系统
指令格式
指令操作码编码
定长编码
变长编码：扩展型编码
指令地址码字段
寻址方式
数据寻址
立即寻址
寄存器寻址
直接寻址
寄存器间接寻址
间接寻址（多重间接）
页面寻址
变址、相对和基址寻址
其他扩展的寻址技术
堆栈
指令寻址
指令分类
CISC
RISC
高级语言与机器级代码之间的对应
编译器、汇编器、链路器基本概念
循环结构语句的机器表示
选择结构语句的机器级表示
过程（函数）调用对应的机器级表示

# 概述

**指令是指示计算机执行某些操作的命令，一台计算机的所有指令的集合构成该机的指令系统，也称指令集。**

指令系统处在软/硬件交界面，同时被硬件设计者和系统程序员看到

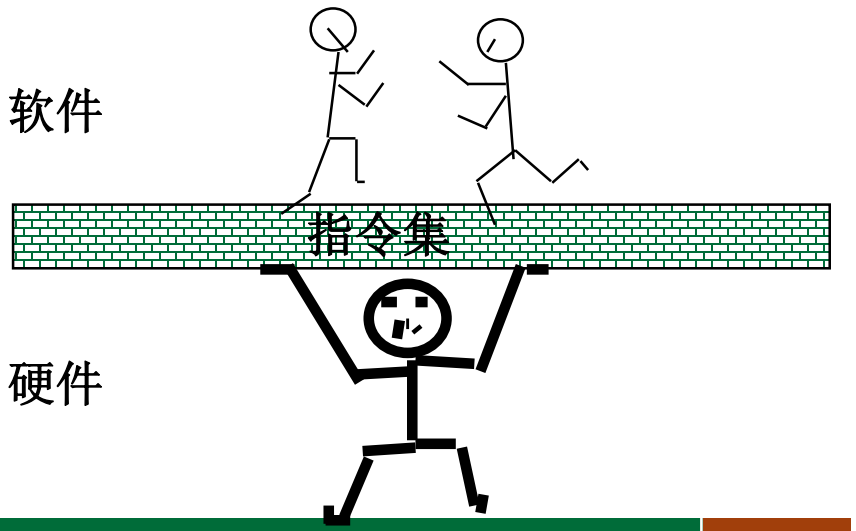
硬件设计者角度：指令系统为CPU提供功能需求，要求易于硬件设计

系统程序员角度：通过指令系统来使用硬件，要求易于编写编译器

指令系统设计的好坏还决定了：计算机的性能和成本。

ISA：（Instruction Set Architecture）一组规范，一般说明寻址方式，指令编码，操作数类型，指令集合等内容。

ISA举例：MIPS，X86（Intel IA-32，Intel 64，AMD64），ARM



# 机器语言与汇编语言



**汇编语言**

- 用助记符表示操作码
- 用标号表示位置用助记符表示寄存器
- .....

汇编器将汇编语言转换为机器语言！

汇编语言编写的优点是：

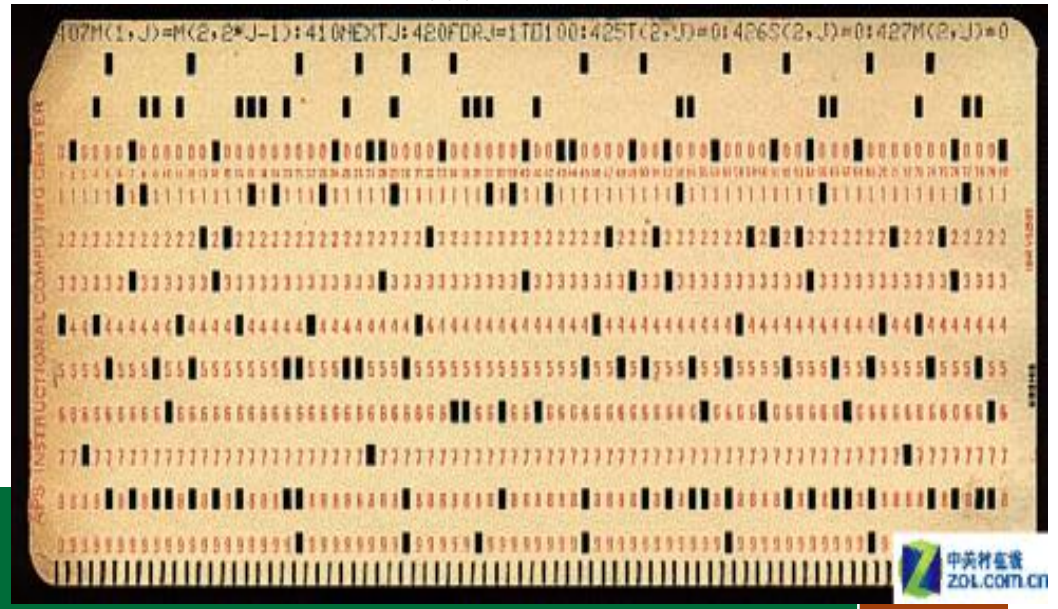
不会因为增减指令而需要修改其他指令

不需记忆指令码，编写方便

可读性比机器语言强

```
;机器语言
8dad0000
240a0001
ad0a0000
ad0a0004
340e0002
...
```

```
#汇编语言
lw      $t5, 0($t5)
sw      $t2, 0($t0)
ori     $t6, $zero, 2
sub     $t1, $t5, $t6
ori     $t7, $zero, 1
...
```



# 高级语言编写程序

```
/*C语言冒泡排序*/
#include<stdio.h>
void main()
{
int a[10];
int i,j,t;
printf("input 10 numbers:\n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);
for(j=0;j<9;j++) /*进行9次循环 实现9趟比较*/
for(i=0;i<9-j;i++) /*在每一趟中进行9-j次比较*/
if(a[i]>a[i+1]) /*相邻两个数升序比较，降序为a[i]<a[i+1]*/
{
t=a[i];
a[i]=a[i+1];
a[i+1]=t;
}
```


**机器语言：**

0011 1100 0001 0000 1010 1011 1100 1101
0010 0110 0001 0001 0000 0000 0000 0010
0010 0110 0001 0010 1111 1111 1111 1110
……

**有两种转换方式：“编译”和“解释”**

**编译程序(Complier)：将高级语言源程序转换为机器级目标程序，执行时只要启动目标程序即可**

**解释程序(Interpreter )：将高级语言语句逐条翻译成机器指令并立即执行，不生成目标文件。**

# 机器指令基本格式



一条指令就是机器语言的一个语句，它是一组有意义的二进制代码。指令的基本格式如下：


**操作码**：指明操作的性质及功能。

**地址码**：指明操作数的地址，特殊情况下也可能直接给出操作数本身。

指令长度应：

① 尽可能短

② 等于字节的整数倍

指令长度可以等于机器字长，也可以大于或小于机器字长。

在一个指令系统中，若所有指令的长度都是相等的，称为定长指令字结构；若各种指令的长度随指令功能而异，称为变长指令字结构。

# 地址码结构



一条**双操作数指令**的除操作码之外，还应包含以下信息：

第一操作数地址，用$A_1$表示；
第二操作数地址，用$A_2$表示；
操作结果存放地址，用$A_3$表示；
下条将要执行指令的地址，用$A_4$表示。

这些信息可以在指令中明显的给出，称为**显地址**；也可以依照某种事先的约定，用隐含的方式给出，称为**隐地址**。

下面以**双操作数指令**为例讨论地址码结构。

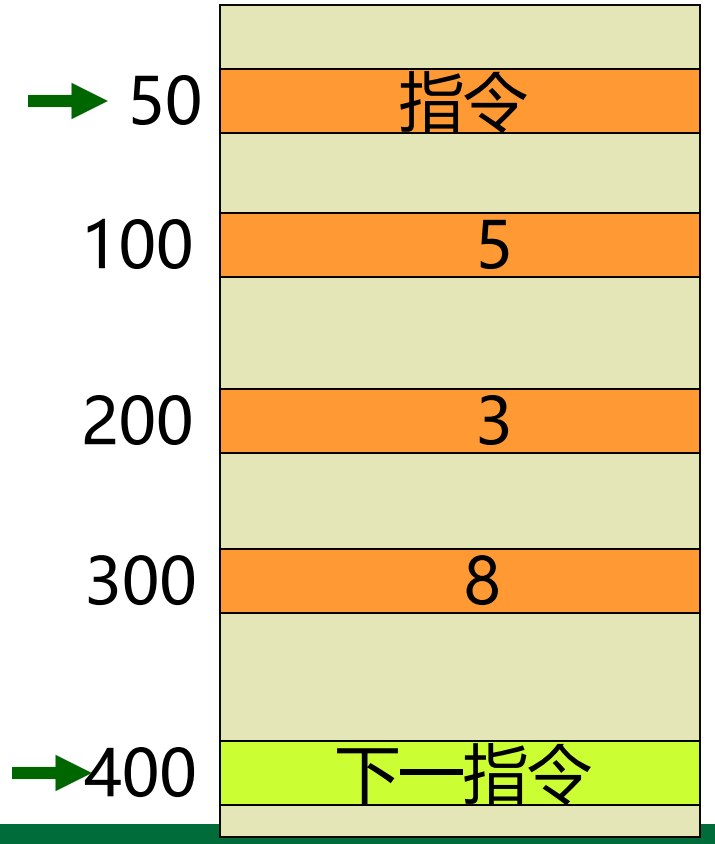
# 四地址指令

| + | 100 | 200 | 300 | 400 |
|---|---|---|---|---|

$(A_1)OP(A_2)\rightarrow A_3$

$A_4$=下条将要执行指令的地址

```
  5
+ 3
-----
  8
```



# 三地址指令

| + | 100 | 200 | 300 |
|---|---|---|---|

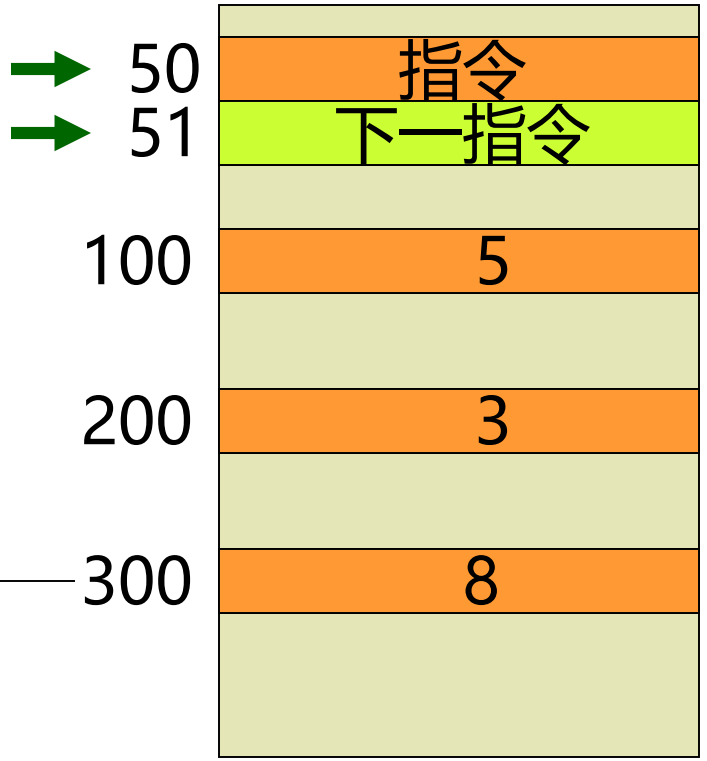
程序计数器：
存放当前指令地址

$(A_1)$ $(A_2)\rightarrow A_3$

(PC)+1=下条将要执行指令的地址

```
    5
+   3
-------
    8
```

执行一条三地址指令需4次访问主存。



# 二地址指令

目的操作数地址

| + | 100 | 200 |
|---|---|---|

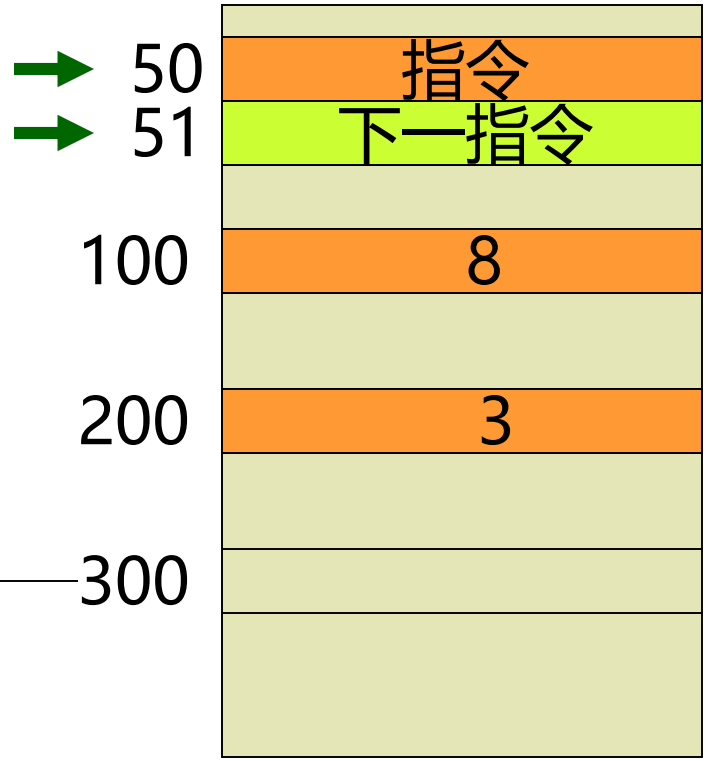
源操作数地址

$(A_1)OP(A_2)\rightarrow A_1$

(PC)+1=下条将要执行指令的地址

$A_1$中原存内容在指令执行后被破坏。

执行一条二地址指令需4次访问主存。




5
+ 3
8

# 一地址指令

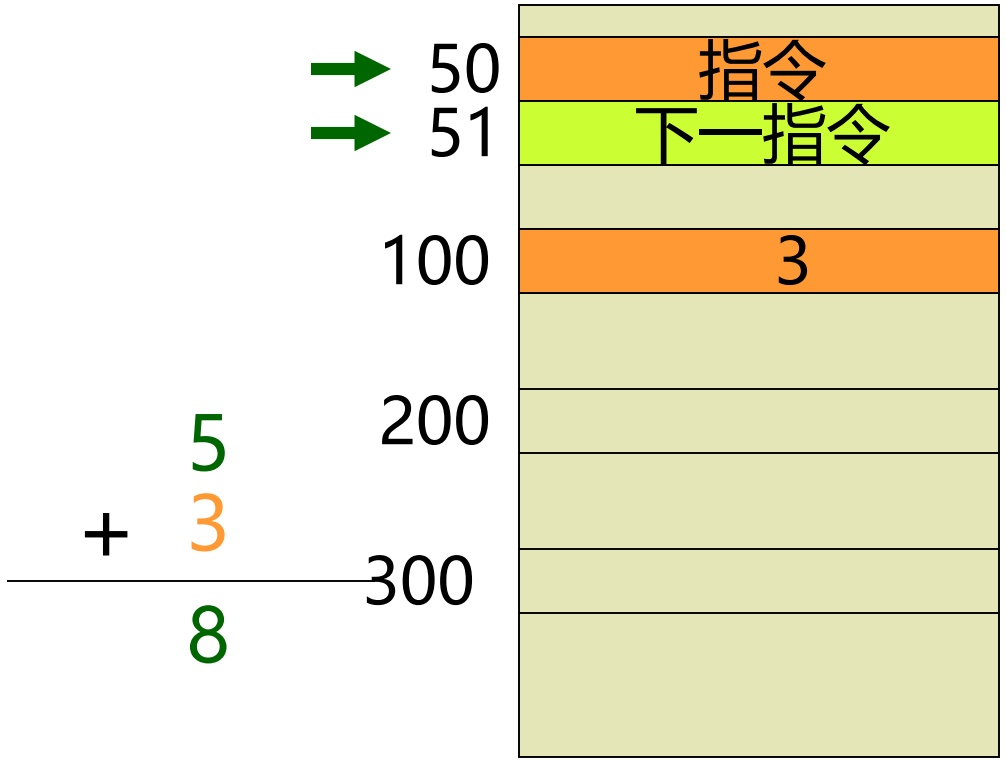
累加寄存器

| + | 100 |
|---|---|

$(A_{cc})OP(A_1)\rightarrow A_{cc}$

(PC)+1=下条将要执行指令的地址

执行一条一地址指令需2次访问主存。



# 零地址指令



OP

操作数地址是隐含的。参加运算的操作数放在堆栈中，运算结果也放在堆栈中。有关堆栈的概念将在稍后讨论。

指令中地址个数的选取要考虑诸多的因素。从缩短程序长度，用户使用方便，增加操作并行度等方面来看，选用三地址指令格式较好；从缩短指令长度，减少访存次数，简化硬件设计等方面来看，一地址指令格式较好。对于同一个问题，用三地址指令编写的程序最短，但指令长度最长，而用二、一、零地址指令来编写程序，程序的长度一个比一个长，但指令的长度一个比一个短。

# 指令的操作码



指令系统中的每一条指令都有一个唯一确定的操作码，指令不同，其操作码的编码也不同。为了能表示整个指令系统中的全部指令，指令的操作码字段应当具有足够的位数。

指令操作码的编码可以分为**规整型**和**非规整型**两类：

规整型（定长编码）

非规整型（变长编码）

# 规整型



操作码字段的位数和位置是固定的。

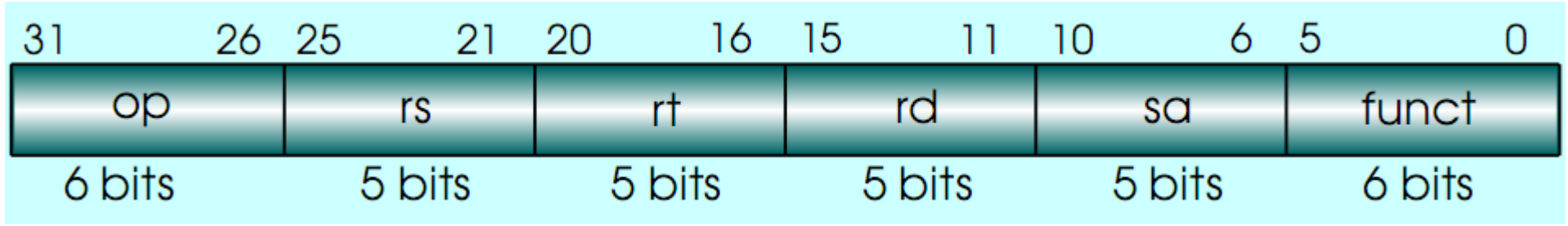
假定：指令系统共有m条指令，指令中操作码字段的位数为N位，则有如下关系式：

$$N \geq \log_2 m$$

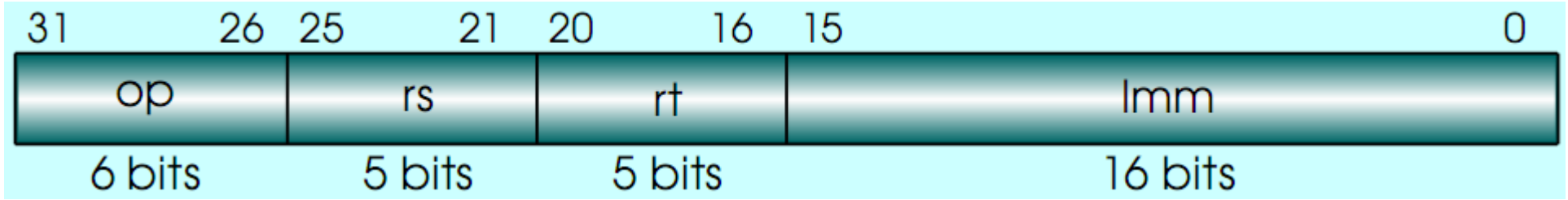
IBM 370机（字长32位）的指令可分为三种不同的长度形式：半字长指令、单字长指令和一个半字长指令。不论指令的长度为多少位，其中操作码字段**一律都是8位**，8位操作码字段允许容纳256条指令，实际上在IBM 370机中仅有183条指令。

# MIPS指令格式

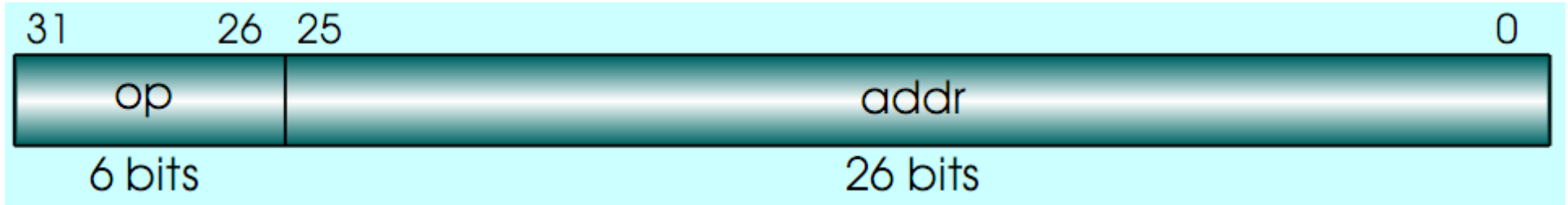
- R型指令（opcode = 6’ b000000）




- I型指令




- J型指令



# R型指令举例



## Assembly Code

```
add $s0, $s1, $s2

sub $t0, $t3, $t5
```

## Field Values

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | 17 | 18 | 16 | 0 | 32 |
| 0 | 11 | 13 | 8 | 0 | 34 |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

## Machine Code

| op | rs | rt | rd | shamt | funct | |
|---|---|---|---|---|---|---|
| 000000 | 10001 | 10010 | 10000 | 00000 | 100000 | (0x02328020) |
| 000000 | 01011 | 01101 | 01000 | 00000 | 100010 | (0x016D4022) |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | |

**MIPS 汇编格式 add rd, rs, rt**

MIPS Architecture For Programmers Volume II-A - The MIPS32 Instruction Set.rev3.02.pdf

# I型指令举例



| Assembly Code | op | rs | rt | imm |
|---|---|---|---|---|
| `addi $s0, $s1, 5` | 8 | 17 | 16 | 5 |
| `addi $t0, $s3, -12` | 8 | 19 | 8 | -12 |
| `lw $t2, 32($0)` | 35 | 0 | 10 | 32 |
| `sw $s1, 4($t1)` | 43 | 9 | 17 | 4 |
| | 6 bits | 5 bits | 5 bits | 16 bits |

汇编格式

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

Machine Code

| op | rs | rt | imm | |
|---|---|---|---|---|
| 001000 | 10001 | 10000 | 0000 0000 0000 0101 | (0x22300005) |
| 001000 | 10011 | 01000 | 1111 1111 1111 0100 | (0x2268FFF4) |
| 100011 | 00000 | 01010 | 0000 0000 0010 0000 | (0x8C0A0020) |
| 101011 | 01001 | 10001 | 0000 0000 0000 0100 | (0xAD310004) |
| 6 bits | 5 bits | 5 bits | 16 bits | |

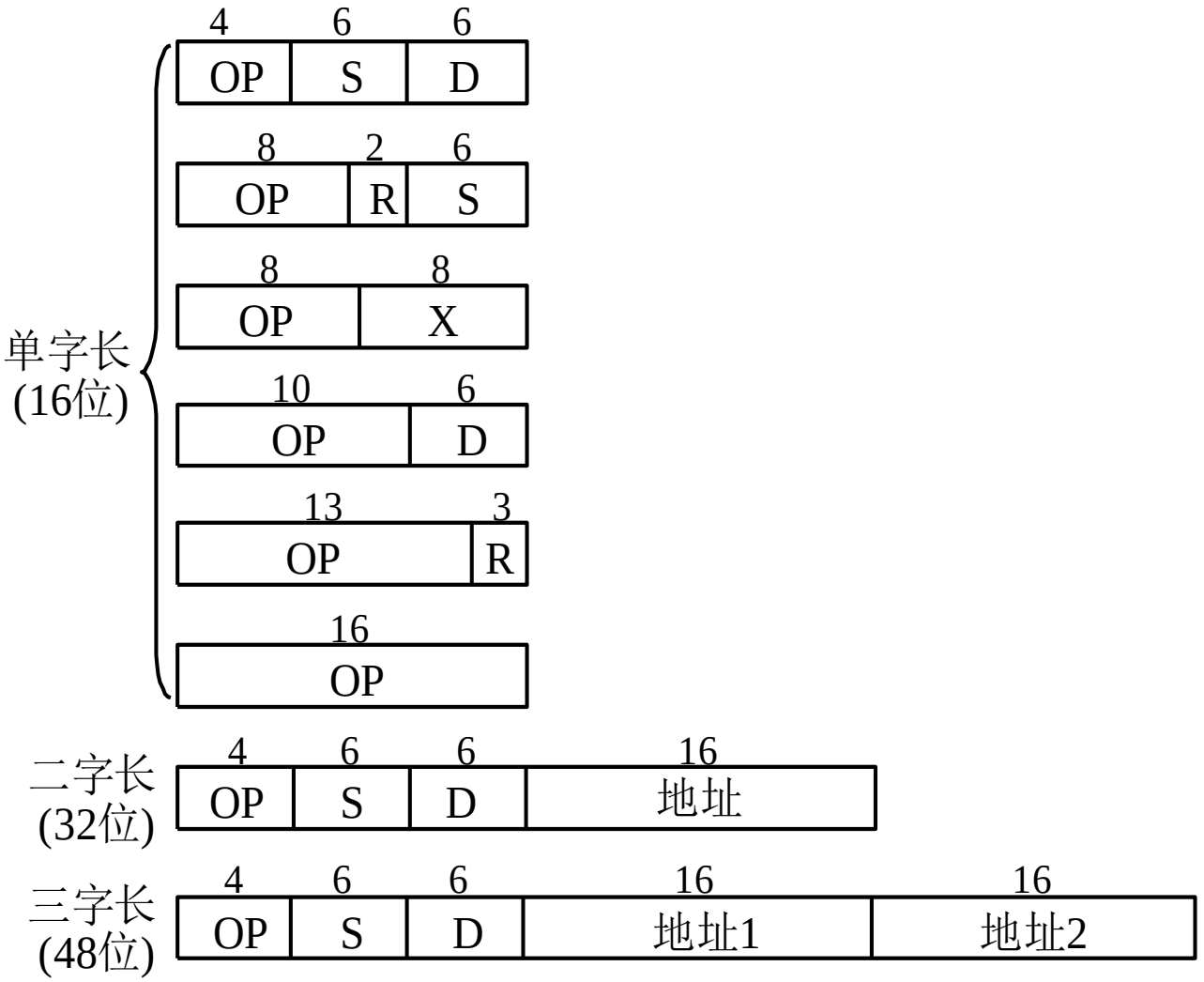
# 非规整型

操作码字段的**位数不固定**，且分散地放在指令字的不同位置上。

PDP-11机（字长16位）的指令分为单字长、两字长、三字长三种，操作码字段占4～16位不等，可遍及整个指令长度。

操作码字段的位数和位置不固定将增加指令译码和分析的难度，使控制器的设计复杂化。

最常用的非规整型编码方式是**扩展操作码法**：让操作数地址个数多的指令（如三地址指令）的操作码字段短些，操作数地址个数少的指令（如一或零地址指令）的操作码字段长些。



# 扩展操作码编码示例

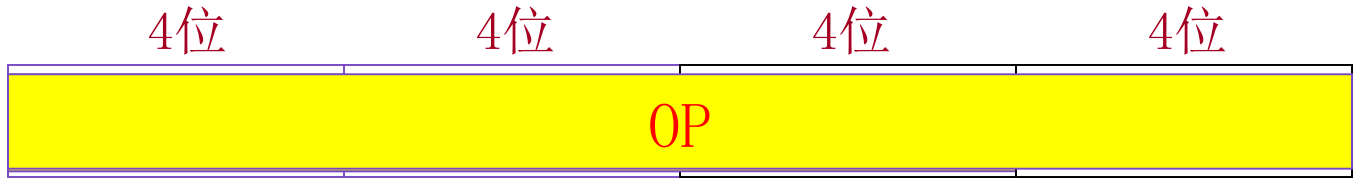


**例如：设某机的指令长度为16位，操作码字段为4位，有三个4位的地址码字段，其格式为：**

| 4位 | 4位 | 4位 | 4位 |
|---|---|---|---|
| OP | $A_1$ | $A_2$ | $A_3$ |

如果按照定长编码的方法，4位操作码字段最多只能表示16条不同的三地址指令。

# 扩展操作码编码示例




**要点：**
**1、短码不能是长码的前缀。**
**2、各条指令的操作码唯一。**

扩展窗口

扩展窗口

扩展窗口

0000 XXXX YYYY ZZZZ
⋮
1110 XXXX YYYY ZZZZ — 15条三地址指令

1111 0000 XXXX YYYY
⋮
1111 1110 XXXX YYYY — 15条二地址指令

1111 1111 0000 XXXX
⋮
1111 1111 1110 XXXX — 15条一地址指令

1111 1111 1111 0000
⋮
1111 1111 1111 1111 — 16条零地址指令

# 指令操作码的优化



指令操作码的优化就是要在足够表达全部指令的前提下，使操作码字段占用的位数最少。操作码优化主要是为了缩短指令字长，减少程序总位数，以节省程序的存储空间。

要对操作码进行优化，就需要知道每种指令在程序中出现的概率（使用频度），每条指令在程序中使用的频度。

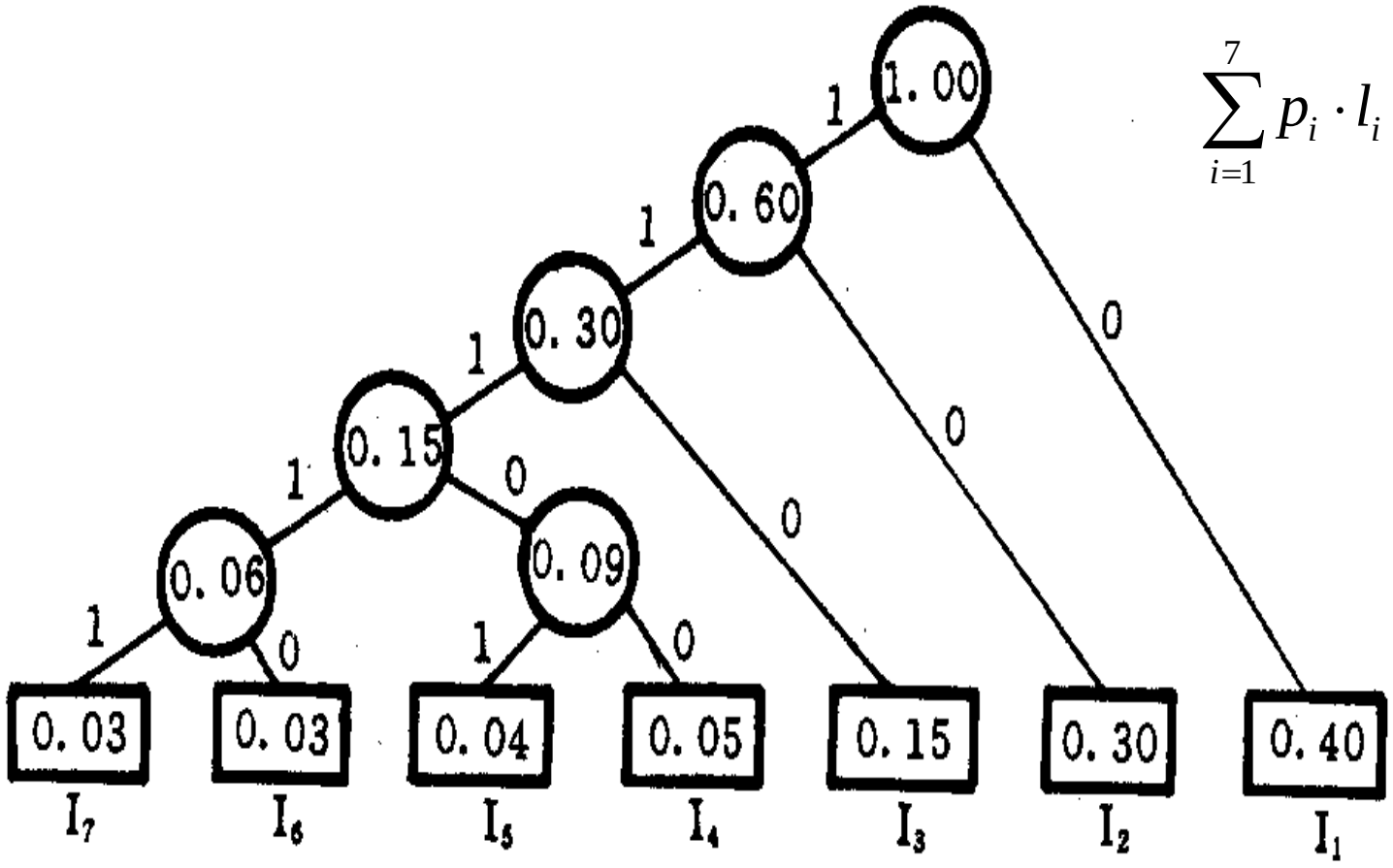
# 指令操作码的优化举例

现有一模型机，共有7条指令，使用频度如下：

| 指令 | 使用频度（$p_i$） |
| --- | --- |
| $I_1$ | 0.40 |
| $I_2$ | 0.30 |
| $I_3$ | 0.15 |
| $I_4$ | 0.05 |
| $I_5$ | 0.04 |
| $I_6$ | 0.03 |
| $I_7$ | 0.03 |

为了减少信息冗余，改用Huffman编码。Huffman编码法编码的原则是：对使用频度较高的指令，分配较短的操作码字段；对使用频度较低的指令，分配较长的操作码字段。Huffman编码首先要构造Huffman树，这种方法又称为最小概率合并法。

# Huffman树




平均码长

$$\sum_{i=1}^{7} p_i \cdot l_i = 0.40\times1+0.30\times2+0.15\times3+0.05\times5$$

$$+0.04\times5+0.03\times5+0.03\times5$$

$$=2.20(\text{位})$$

# Huffman编码法

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

| 指令序号 | 概率 | Huffman编码法 | 操作码长度 |
|---|---|---|---|
| $I_1$ | 0.40 | 0 | 1位 |
| $I_2$ | 0.30 | 10 | 2位 |
| $I_3$ | 0.15 | 110 | 3位 |
| $I_4$ | 0.05 | 11100 | 5位 |
| $I_5$ | 0.04 | 11101 | 5位 |
| $I_6$ | 0.03 | 11110 | 5位 |
| $I_7$ | 0.03 | 11111 | 5位 |

- Huffman编码是最优化的编码，但这种编码存在着的主要缺点有：
- ① 操作码长度很不规整，硬件译码困难。
- ② 与地址码共同组成固定长的指令比较困难。

扩展编码法

由固定长操作码与Huffman编码法相结合形成的一种编码方式，操作码长度被限定使用有限的几种码长，仍体现高概率指令用短码，低概率指令用长码的思想，使操作码的平均码长虽大于Huffman编码，但小于等长编码，是一种实际可用的优化编码方法。

- 例如：将上例改为2-4等长扩展编码。

| 指令序号 | 概率 | 2 - 4扩展操作码 | 操作码长度 |
|---|---|---|---|
| $I_1$ | 0.40 | 00 | 2位 |
| $I_2$ | 0.30 | 01 | 2位 |
| $I_3$ | 0.15 | 10 | 2位 |
| $I_4$ | 0.05 | 1100 | 4位 |
| $I_5$ | 0.04 | 1101 | 4位 |
| $I_6$ | 0.03 | 1110 | 4位 |
| $I_7$ | 0.03 | 1111 | 4位 |

# 寻址技术



寻址，指的是寻找操作数的地址或下一条将要执行的指令地址。寻址技术包括编址方式和寻址方式。

## 编址

通常，指令中的地址码字段将指出操作数的来源和去向，而操作数则存放在相应的存储设备中。在计算机中需要编址的设备主要有CPU中的通用寄存器、主存储器和输入输出设备等3种。

## 编址单位

(1)字编址

每个编址单位所包含的信息量（二进制位数）与读或写一次寄存器、主存所获得的信息量是相同的。早期的大多数机器都采用这种编址方式。

(2)字节编址

字节编址是为了适应非数值计算的需要。字节编址方式使编址单位与信息的基本单位（一个字节）相一致，这是它的最大优点。然而，如果主存的访问单位也是一个字节的话，那么主存的频带就太窄了。

编址单位 < 访问单位

通常主存的访问单位是编址单位的若干倍。

(3)位编址

也有部分计算机系统采用位编址方式。

## 3. 指令中地址码的位数

指令格式中每个地址码的位数是与**主存容量**和**最小寻址单位**（即编址单位）有关联的。主存容量越大，所需的地址码位数就越长。对于相同容量来说，如果以字节为最小寻址单位，地址码的位数就需要长些；如果以字为最小寻址单位（假定字长为16位或更长），地址码的位数可以减少。设某机主存容量为$2^{20}$个字节，机器字长32位。若最小寻址单位为字节（按字节编址），其地址码应为**20**位；若最小寻址单位为字（按字编址），其地址码只需**18**位。

# 数据寻址和指令寻址



寻址可以分为**指令寻址和数据寻址**。

寻找下一条将要执行的指令地址称为**指令寻址**，指令寻址比较简单，它又可以细分为顺序寻址和跳跃寻址。

**顺序寻址**可通过程序计数器PC加1，自动形成下一条指令的地址；跳跃寻址则需要通过程序转移类指令实现。

**跳跃寻址**的转移地址形成方式有三种：直接（绝对）、相对和间接寻址，它与下面介绍的数据寻址方式中的直接、相对和间接寻址是相同的，只不过寻找到的不是操作数的有效地址而是转移的有效地址而已。

寻找操作数的地址称为**数据寻址**，**数据寻址方式较多，其最终目的都是寻找所需要的操作数**。

# 基本的数据寻址方式



寻址方式是根据指令中给出的地址码字段寻找真实操作数地址的方式。

指令中的形式地址A ——寻址方式——→ 有效地址EA

## 1.立即寻址

| OP | 立即数 |
|---|---|

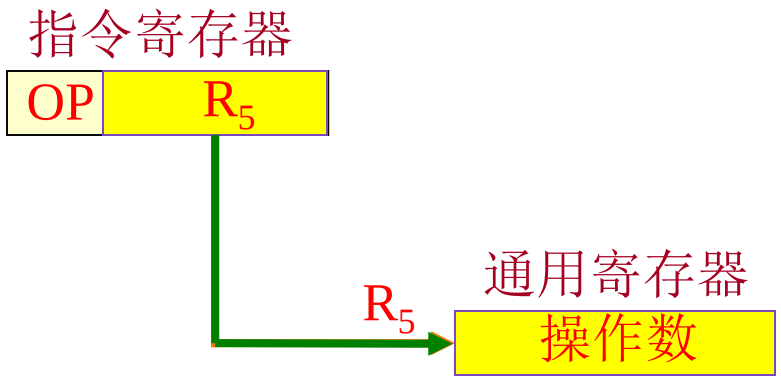
在取指令时，操作码和操作数被同时取出，不必再次访问存储器，从而提高了指令的执行速度。但是，因为操作数是指令的一部分，不能被修改，且立即数的大小将受到指令长度的限制。

## 2.寄存器寻址

指令中地址码部分给出某一通用寄存器的编号，所指定的寄存器中存放着操作数。

两个明显的优点：

- 从寄存器存取数据比主存快得多；
- 由于寄存器的数量较少，其地址码字段比主存单元地址字段短得多。




$EA = R_i$

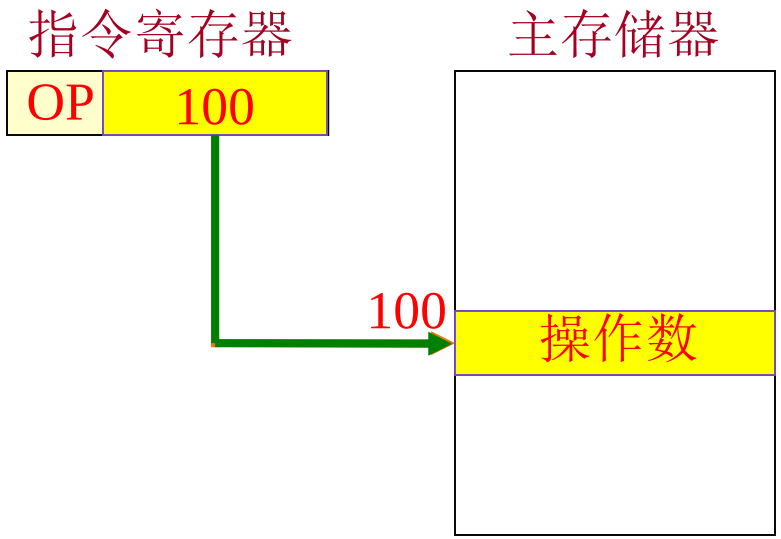
操作数$S = (R_i)$

## 3.直接寻址

**指令中地址码字段给出的地址A就是操作数的有效地址：**

**EA=A**

**由于操作数地址是不能修改的，与程序本身所在的位置无关，所以又叫做绝对寻址方式。**




操作数S=(A)

## 4.间接寻址

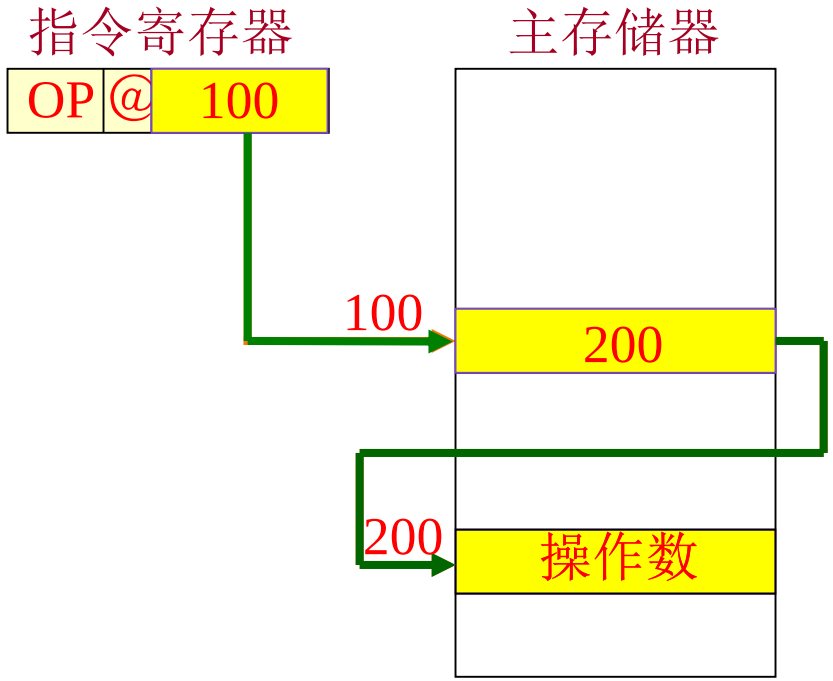
指令中给出的地址A不是操作数的地址，而是存放操作数地址的地址。

EA=(A)

通常在指令格式中划出一位@作为标志位。

@=0 直接寻址

@=1 间接寻址




操作数S=((A))

间接寻址要比直接寻址灵活得多，它的**主要优点为**：

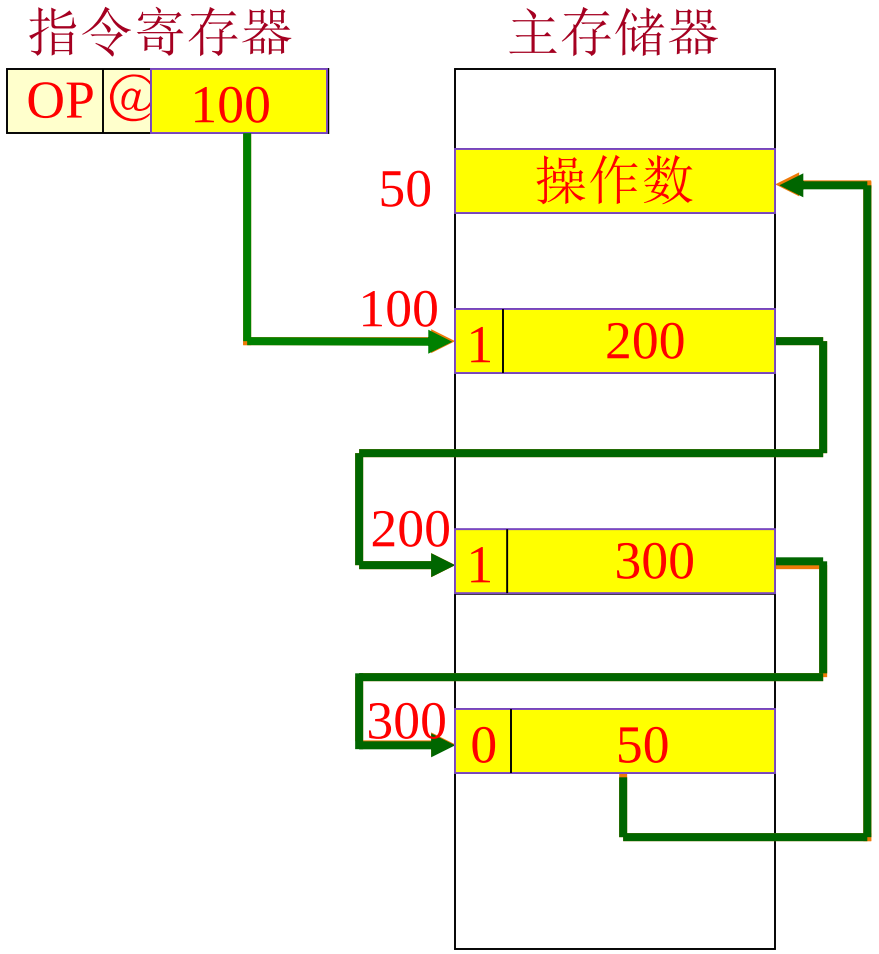
- 扩大了寻址范围，可用指令的短地址访问大的主存空间。
- 可将主存单元作为程序的地址指针，用以指示操作数在主存中的位置。当操作数的地址需要改变时，不必修改指令，只需修改存放有效地址的那个主存单元（间接地址单元）的内容。

除去一级间接寻址外，还有多级间接寻址。多级间接寻址为取得操作数需要多次访问主存，即使在找到操作数有效地址后，还需再访问一次主存才可得到真正的操作数。

多级间接标志：

0：找到有效地址

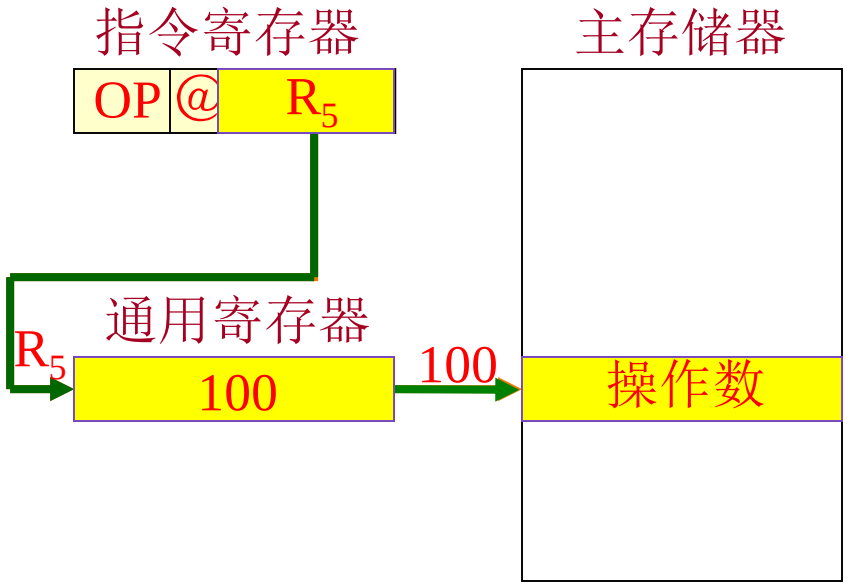
1：继续间接寻址



## 5.寄存器间接寻址

指令中的地址码给出某一通用寄存器的编号，被指定的寄存器中存放操作数的有效地址，而操作数则存放在主存单元中。

这种寻址方式的指令较短，并且在取指后只需一次访存便可得到操作数。




$EA=(R_i)$

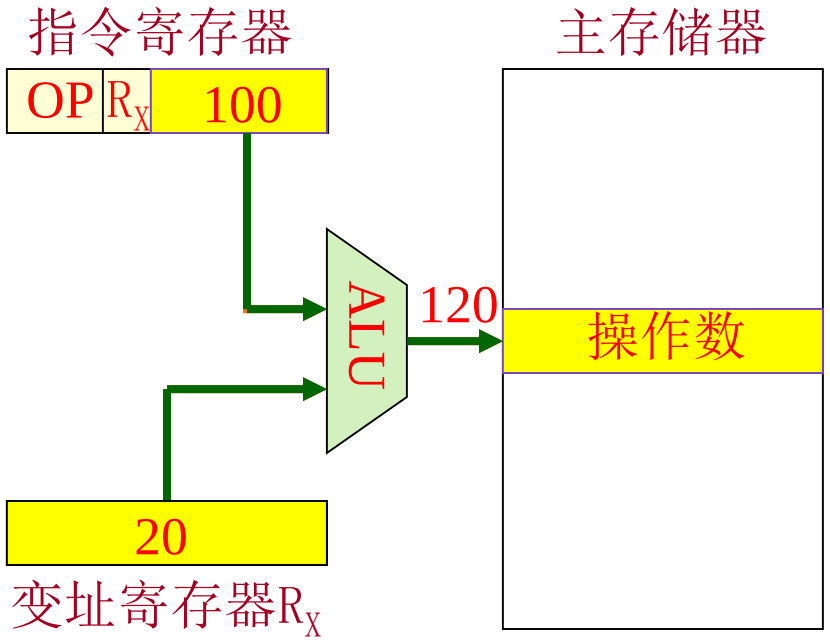
操作数$S=((R_i))$

## 6.变址寻址

把指令给出的形式地址A与变址寄存器$R_X$的内容相加，形成操作数有效地址：

$$EA=A+(R_X)$$

$R_X$ 的内容为变址值。




变址寻址是一种广泛采用的寻址方式，通常指令中的形式地址作为基准地址，而$R_X$的内容作为修改量。在遇到需要频繁修改地址时，无须修改指令，只要修改变址值就可以了。

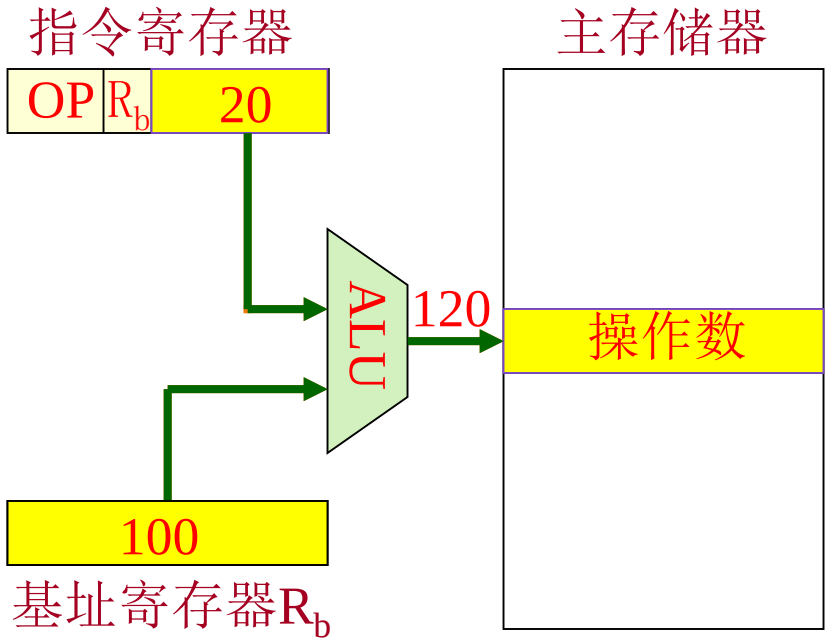
操作数$S=(A+(R_X))$

## 7.基址寻址

将基址寄存器$R_b$的内容与位移量D相加，形成操作数有效地址：

$$EA=(R_b)+D$$

基址寄存器的内容称为基址值，指令的地址码字段是一个位移量，**位移量可正可负**。




操作数$S=((R_b)+D)$

# 基址寻址与变址寻址的区别



基址寻址和变址寻址在形成有效地址时所用的算法是相同的，而且在一些计算机中，这两种寻址方式都是由同样的硬件来实现的。

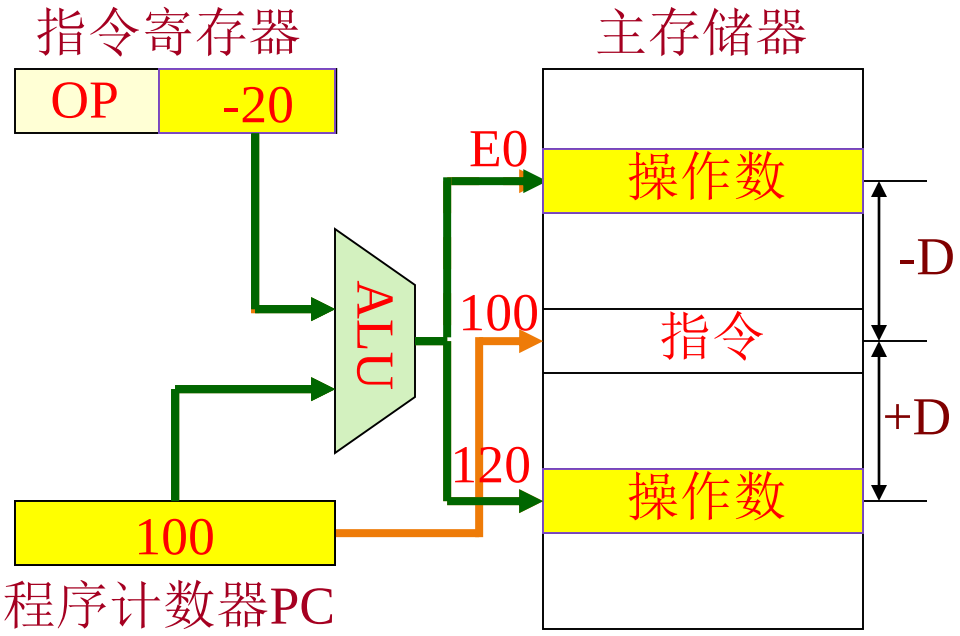
但这两种寻址方式应用的场合不同，变址寻址是面向用户的，用于访问字符串、向量和数组等成批数据；而基址寻址面向系统，主要用于逻辑地址和物理地址的变换，用以解决程序在主存中的再定位和扩大寻址空间等问题。在某些大型机中，基址寄存器只能由特权指令来管理，用户指令无权操作和修改。

## 8.相对寻址

相对寻址是基址寻址的一种变通，由程序计数器PC提供基准地址，即：

$$EA=(PC)+D$$

位移量指出的是操作数和现行指令之间的相对位置。



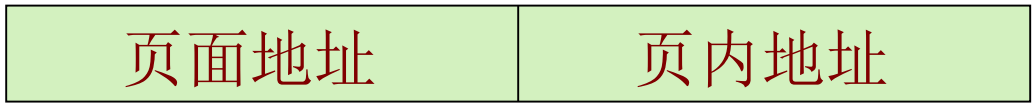

操作数S=((PC)+D)

**相对寻址方式的特点：**

- **操作数的地址不是固定的，它随着PC值的变化而变化，并且与指令地址之间总是相差一个固定值±D。当指令地址改变时，由于其位移量不变，使得操作数与指令在可用的存储区内一起移动，所以仍能保证程序的正确执行。采用PC相对寻址方式编写的程序可在主存中任意浮动，它放在主存的任何地方，所执行的效果都是一样的。**
- **由于指令中给出的位移量可正、可负，所以对于指令地址而言，操作数地址可能在指令地址之前或之后。**

## 9.页面寻址

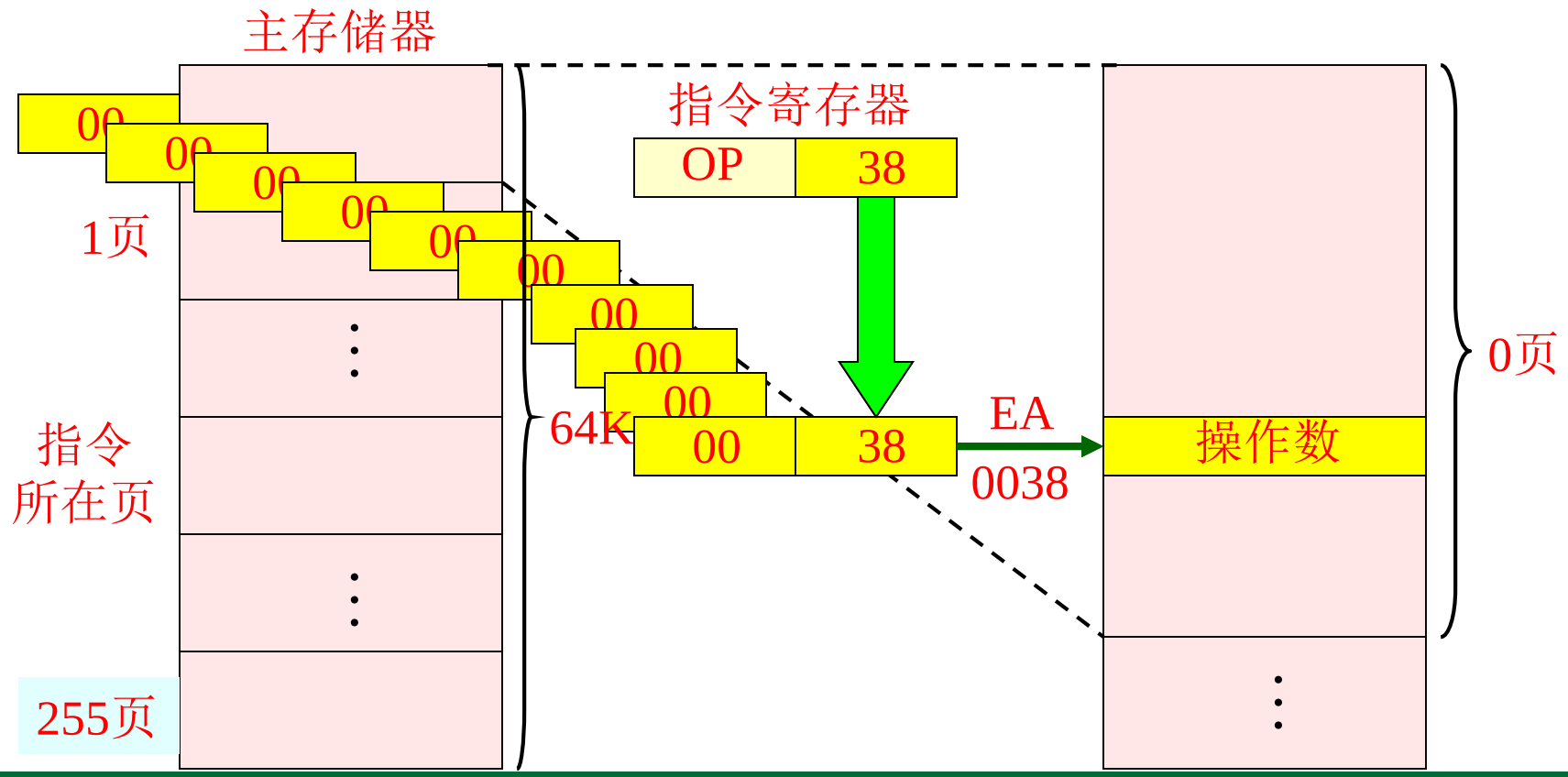
页面寻址相当于将整个主存空间分成若干个大小相同的区，每个区称为一页，每页有若干个主存单元。每页都有自己的编号，称为页面地址；页面内的每个主存单元也有自己的编号，称为页内地址。这样，操作数的有效地址就被分为两部分：前部为页面地址，后部为页内地址。



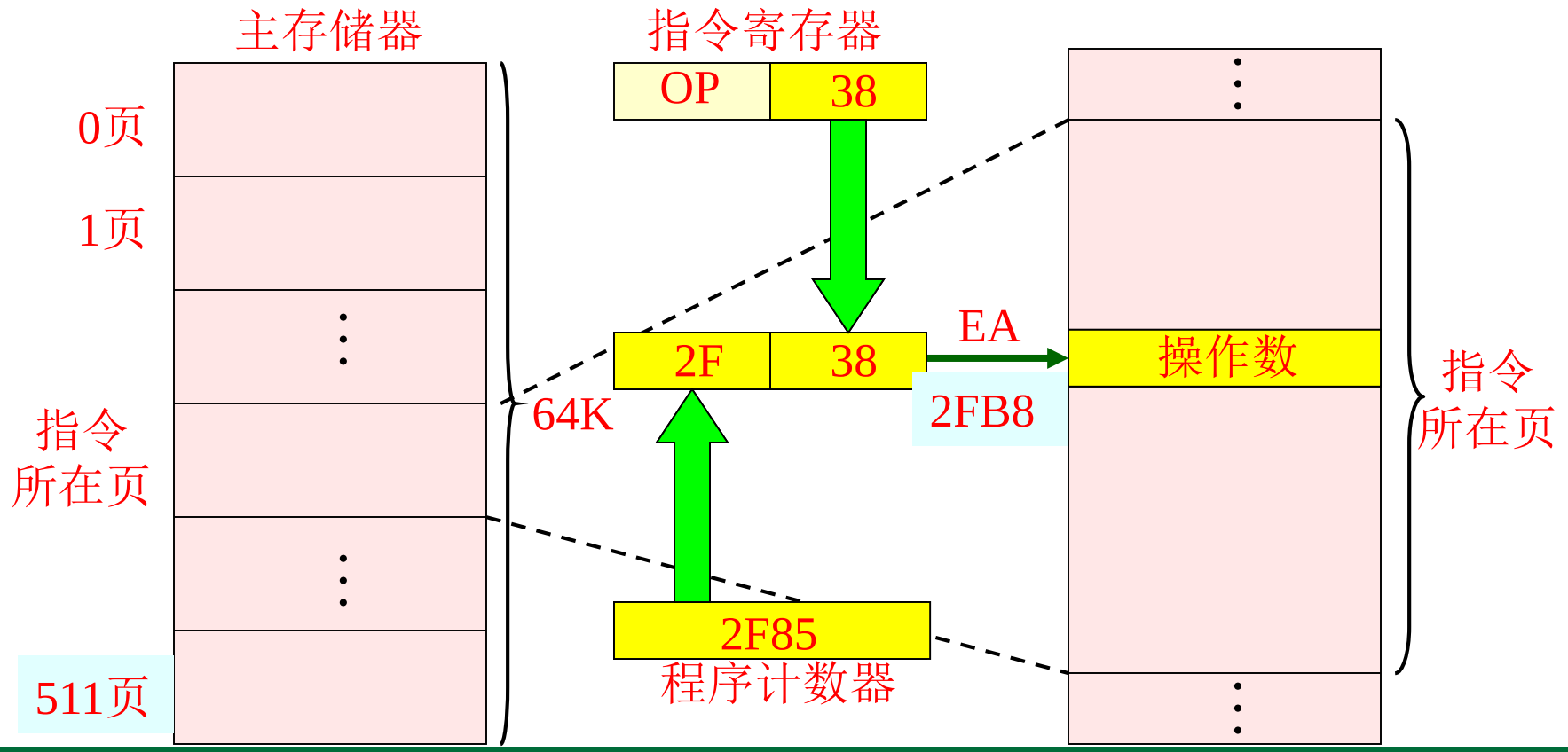

| 页面地址(7位) | 页内地址(9位) |
|---|---|

**页内地址由指令的地址码A自动直接提供，它与页面地址通过简单的拼装连接就可得到有效地址。根据页面地址的来源不同，有三种不同的页面寻址：**

**(1)基页寻址（零页寻址）。由于页面地址等于全0，所以有效地址EA=0//A（//在这里表示简单拼接），操作数S在零页面中。**



**(2)当前页寻址。页面地址就等于程序计数器PC的高位部分，所以有效地址$EA=(PC)_H \ // \ A$，操作数S与指令本身处于同一页面中。**



(3)页寄存器寻址。页面地址取自页寄存器，与形式地址相拼接形成操作数有效地址：

EA=(页寄存器)//A

有些计算机在指令格式中设置了一个页面标志位（Z/C）。当Z/C=0，表示零页寻址，当Z/C=1，表示当前页寻址。

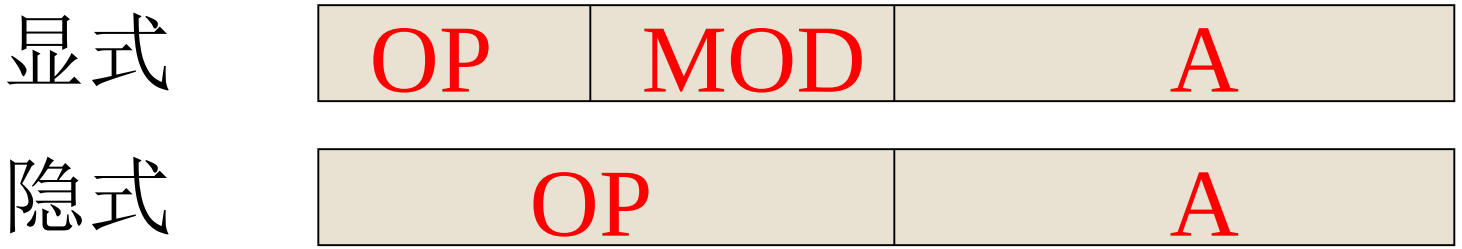
# 数据寻址方式汇总

- 各种数据寻址方式获得数据的速度（由快到慢）
  - 立即寻址
  - 寄存器寻址
  - 直接寻址
  - 寄存器间接寻址
  - 页面寻址
  - 变址寻址（基址寻址、相对寻址）
  - 一级间接寻址
  - 多级间接寻址

为了能区分出各种不同寻址方式，必须在指令中给出标识。标识的方式通常有两种：**显式和隐式**。

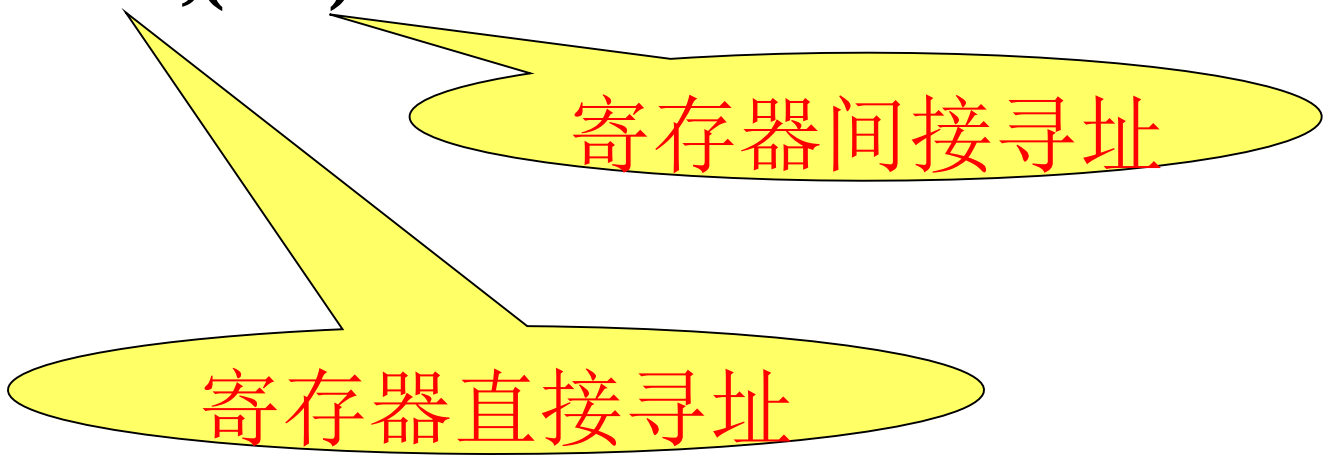
显式的方法就是在指令中设置专门的寻址方式（MOD）字段，用二进制代码来表明寻址方式类型。

隐式的方式是由指令的操作码字段说明指令格式并隐含约定寻址方式。



注意，一条指令若有两个或两个以上的地址码时，**各地址码可采用不同的寻址方式**。例如，源地址采用一种寻址方式，而目的地址采用另一种寻址方式。

MOV AX,(BX)



# 堆栈与堆栈操作



堆栈是一种按特定顺序进行存取的存储区，这种特定顺序可归结为“后进先出”（LIFO）或“先进后出”（FILO）。

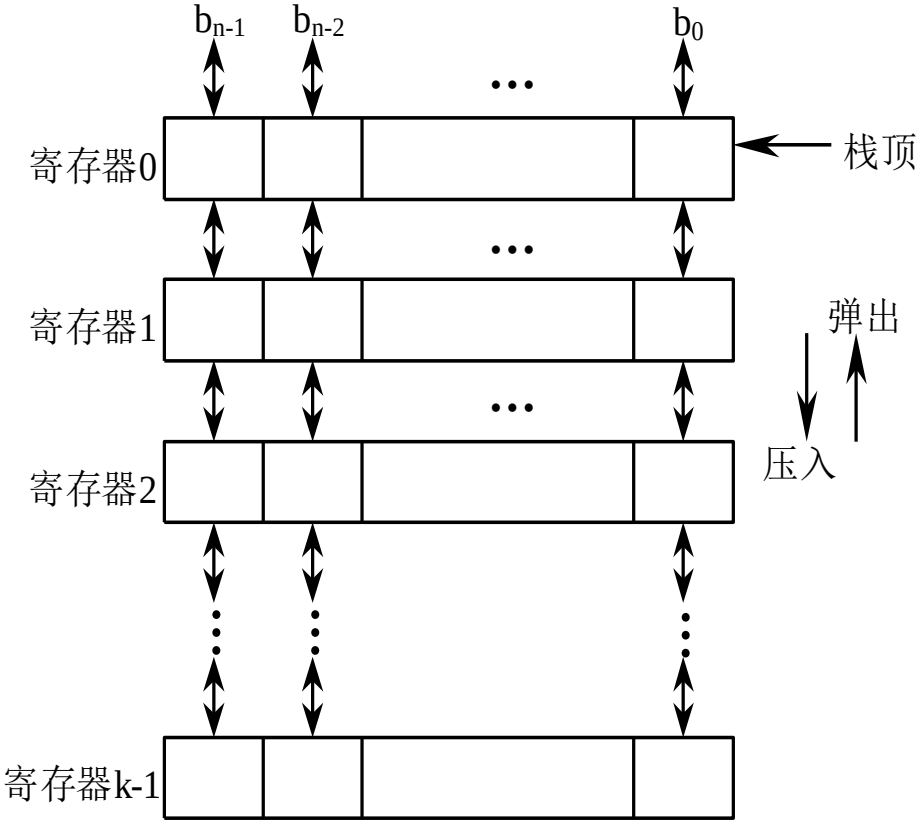
堆栈主要用来暂存中断断点、子程序调用时的返回地址、状态标志及现场信息等，也可用于子程序调用时参数的传递，所以用于访问堆栈的指令只有进栈（压入）和出栈（弹出）两种。

# 堆栈类型



## 1.寄存器堆栈

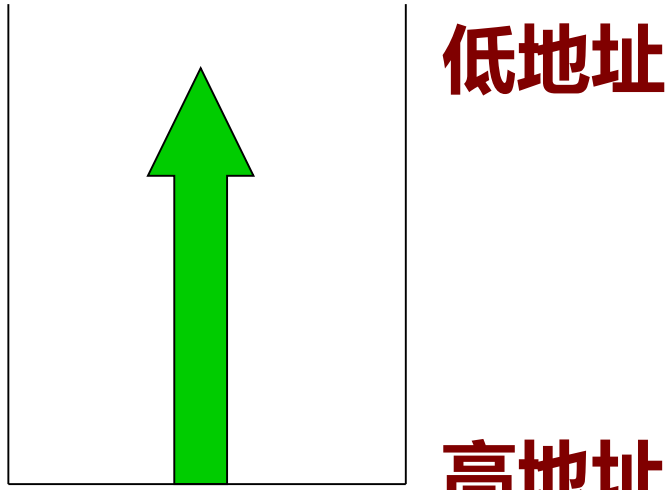
用一组专门的寄存器构成寄存器堆栈，又称为硬堆栈。这种堆栈的栈顶是固定的，寄存器组中各寄存器是相互连接的，它们之间具有对应位自动推移的功能，即可将一个寄存器的内容推移到相邻的另一个寄存器中去。



# 2.存储器堆栈

从主存中划出一段区域来作堆栈，这种堆栈又称为软堆栈，堆栈的大小可变，栈底固定，栈顶浮动，故需要一个专门的硬件寄存器作为堆栈栈顶指针SP，简称栈指针。栈指针所指定的主存单元，就是堆栈的栈顶。

自底向上生成方式的堆栈



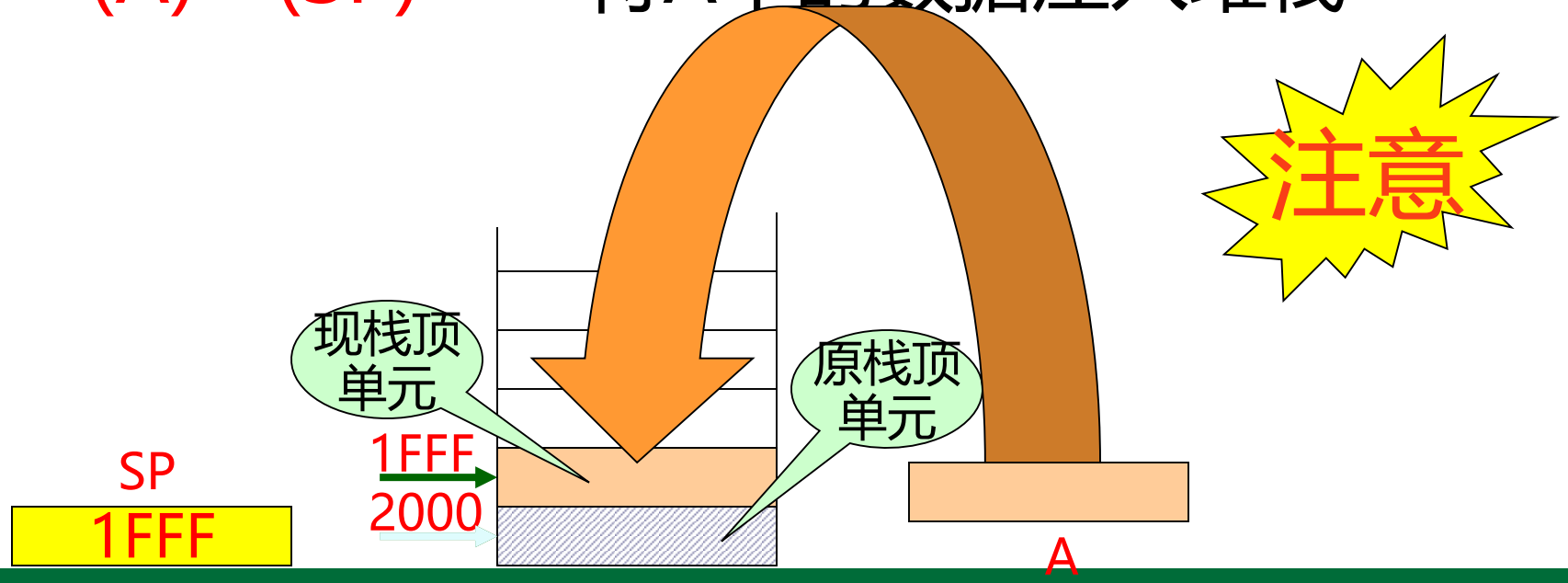
# 堆栈与堆栈操作



堆栈的栈底地址大于栈顶地址，通常栈指针始终指向**栈顶的满单元**。进栈时，SP的内容需要先自动减1，然后再将数据压入堆栈。

(SP)-1→SP　　修改栈指针

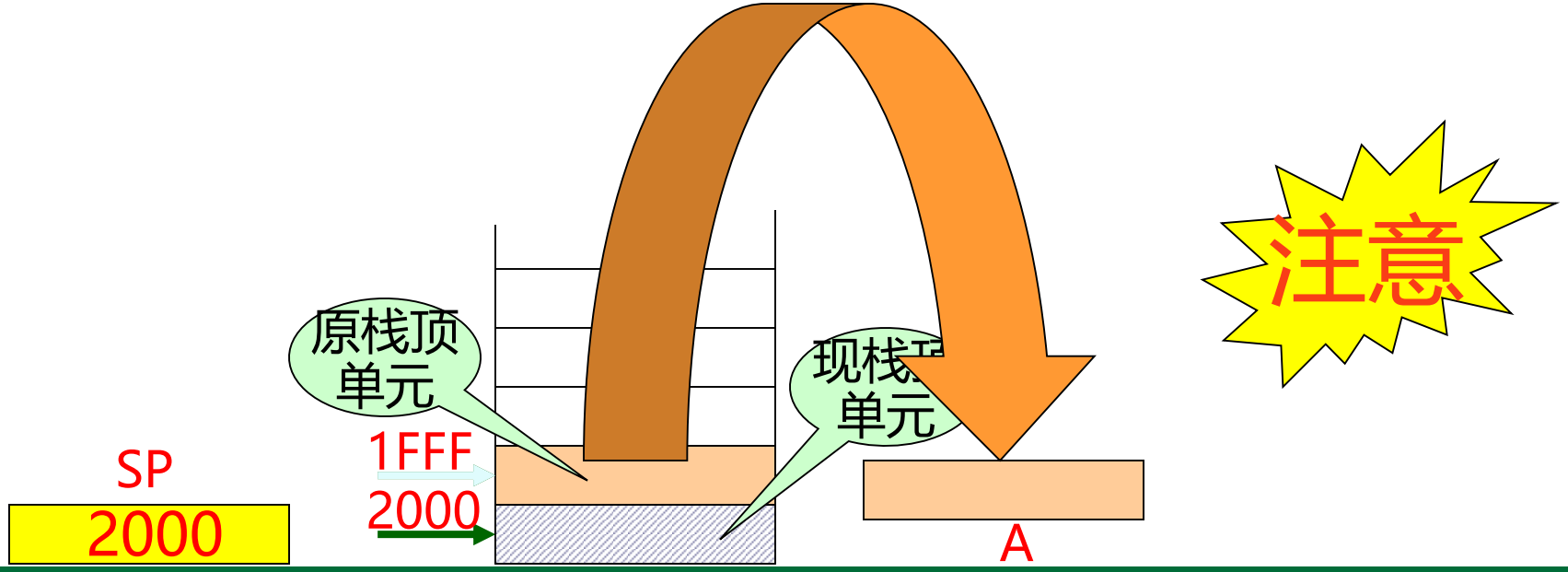
(A)→(SP)　　将A中的数据压入堆栈



**出栈时，需要先将堆栈中的数据弹出，然后SP的内容再自动加1。**

((SP))→A　　将栈顶内容弹出，送入A中

(SP)+1→SP　　修改栈指针



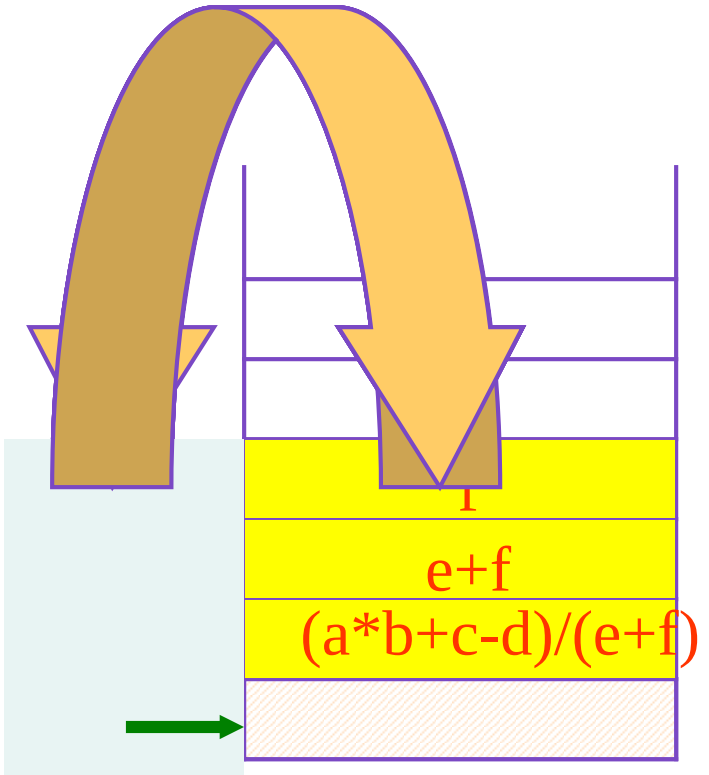
## 堆栈计算机实现计算x=(a×b+c-d) ÷(e+f)

```
PUSH    A
PUSH    B
MUL
PUSH    C
ADD
PUSH    D
SUB
PUSH    E
PUSH    F
ADD
DIV
POP     X
```

12条指令，进、出栈指令3次访存，算逻指令4次访存，执行此程序共访存
7×3+5×4=41次。

## 可以先把算术表达式转换成逆波兰表示式：

x=ab×c+d-ef+÷



## 数据传送类指令

数据传送类指令主要用于实现寄存器与寄存器之间，寄存器与主存单元之间以及两个主存单元之间的数据传送。数据传送类指令又可以细分为：

### 1.一般传送指令

一般传送指令（MOV）具有数据复制的性质，即数据从源地址传送到目的地址，而源地址中的内容保持不变。传送通常以字节、字、双字或数组为单位，特殊情况下也能按位为单位进行传送。

# 数据传送类指令



(1)主存单元之间的传送

MOV $mem_2$,$mem_1$，其含义为 $(mem_1)\rightarrow mem_2$

(2)从主存单元传送到寄存器

MOV reg,mem，其含义为 (mem)→reg

在有些计算机中，该指令用助记符LOAD表示，又称为取数指令。

(3)从寄存器传送到主存单元

MOV mem,reg，其含义为 (reg)→mem

在有些计算机里，该指令用助记符STORE表示，又称为存数指令。

(4)寄存器之间的传送

MOV $reg_2$,$reg_1$，其含义为 $(reg_1)\rightarrow reg_2$

# 数据传送类指令



**2.堆栈操作指令**

**堆栈指令是一种特殊的数据传送指令，分为进栈（PUSH）和出栈（POP）两种。**

**因为堆栈（指软堆栈）是主存的一个特定区域，所以对堆栈的操作也就是对存储器的操作。**

**3.数据交换指令**

**前述指令的传送都是单方向的，然而，数据传送也可以是双方向的，即将源操作数与目的操作数（一个字节或一个字）相互交换位置。**

# 运算类指令



## 1.算术运算指令

算术运算指令主要用于进行定点和浮点运算。这类运算包括加、减、乘、除以及加1、减1、比较等，有些机器还有十进制算术运算指令。

绝大多数算术运算指令都会影响到状态标志位，通常的标志位有进位、溢出、全零、正负和奇偶等。

## 2.逻辑运算指令

一般计算机都具有与、或、非、异或等逻辑运算指令。这类指令在没有设置专门的位操作指令的计算机中常用于对数据字（字节）中某些位（一位或多位）进行操作。

# 逻辑运算类指令举例

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

**(1)按位测（位检查）**

$$\begin{array}{l} \quad XXXXXXXX \\ \wedge \quad 00010000 \\ \hline \quad 000X0000 \end{array}$$

**(2)按位修改**

**利用“异或”指令可以修改目的操作数的某些位，只要源操作数的相应位为“1”，其余位为“0”，异或之后就达到了修改这些位的目的（因为A⊕1=A，A⊕0=A）。**

$$\begin{array}{l} \quad XXXXXXXX \\ \oplus \quad 00001000 \\ \hline \quad XXXX\overline{X}XXX \end{array}$$

## 3.移位指令

分为算术移位、逻辑移位和循环移位3类，它们又可分为左移和右移两种。

算术移位的对象是带符号数，算术移位过程中必须保持操作数的符号不变，左移一位，数值×2，右移一位，数值÷2。

逻辑移位的对象是没有数值含义的二进制代码，因此移位时不必考虑符号问题。

循环移位又按进位位是否一起循环分为两类：

- 小循环（不带进位循环）
- 大循环（带进位循环）

# 程序控制类指令



程序控制类指令用于控制程序的执行方向，并使程序具有测试、分析与判断的能力。

1.转移指令

在程序执行过程中，通常采用转移指令来改变程序的执行方向。转移指令又分无条件转移和条件转移两种。

无条件转移指令（JMP）不受任何条件的约束，直接把程序转向新的位置执行。

条件转移指令必须受到条件的约束，若条件满足时才转向新的位置执行，否则程序仍顺序执行。

# 子程序调用指令

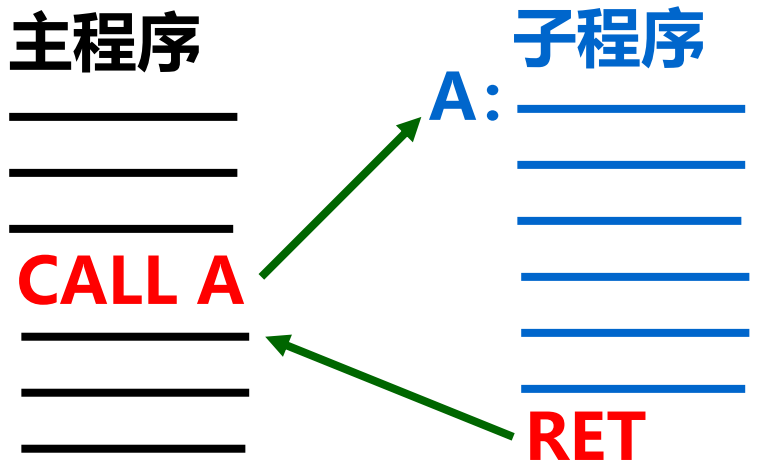


**子程序是一组可以公用的指令序列，只要知道子程序的入口地址就能调用它。**

从主程序转向子程序的指令称为子程序调用指令（CALL）；而从子程序转向主程序的指令称为返回指令（RET）。

主程序和子程序是相对的概念，调用其它程序的程序是主程序；被其它程序调用的程序是子程序。

转子指令安排在主程序中需要调用子程序的地方，**转子指令是一地址指令**。



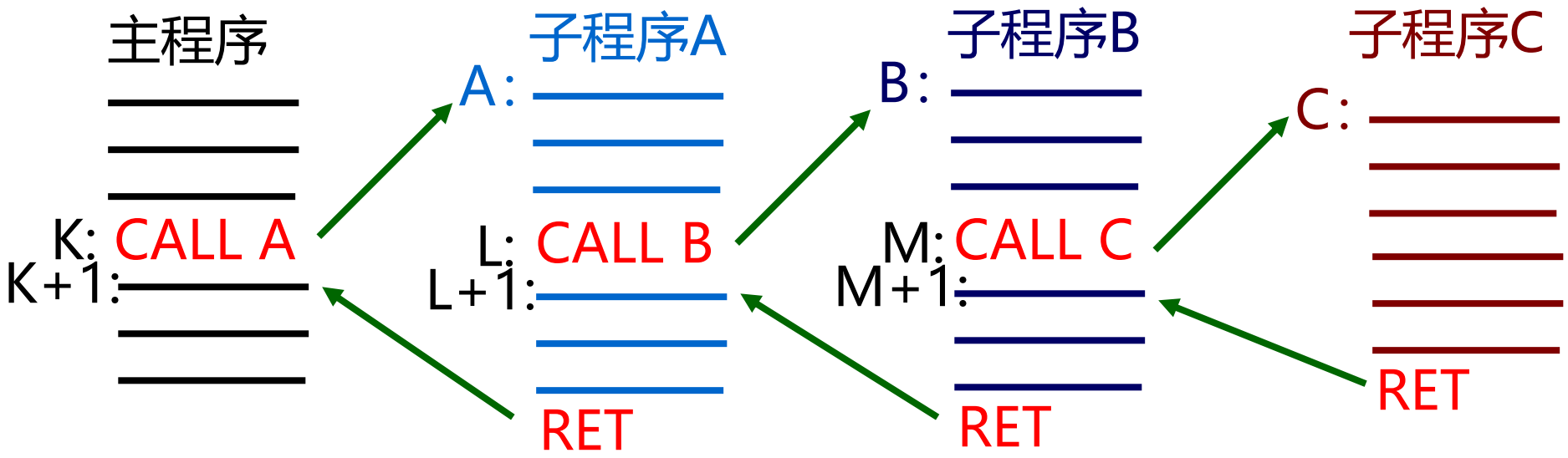
# 子程序调用和转移指令区别

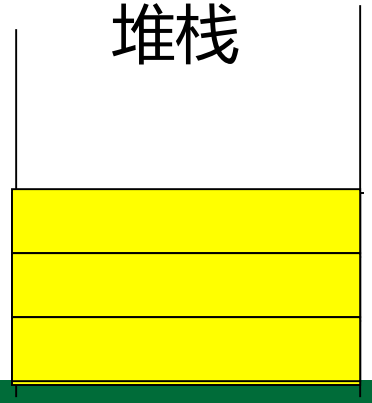


子程序调用指令和转移指令都可以改变程序的执行顺序，但两者存在着很大的差别：

- 转移指令转移到指令中给出的转移地址处执行指令，不存在返回要求，没有返回地址问题；而子程序调用指令必须以某种方式保存返回地址，以便返回时能找到原来的位置。
- 转移指令用于实现同一程序内的转移；而子程序调用指令转去执行一段子程序，实现的是程序与程序之间的转移。

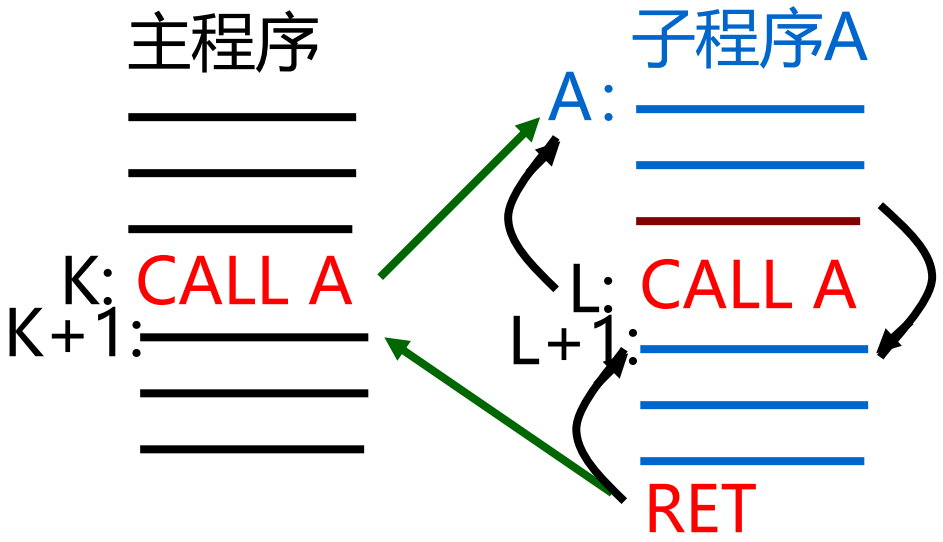
# 子程序嵌套

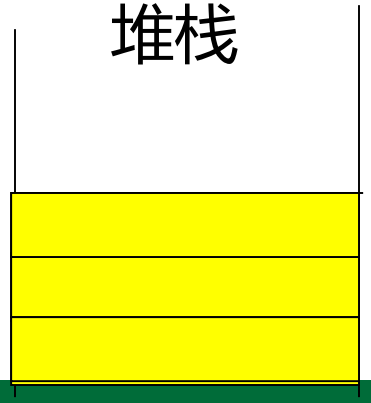



利用堆栈保存返回地址

# 子程序递归




利用堆栈保存返回地址

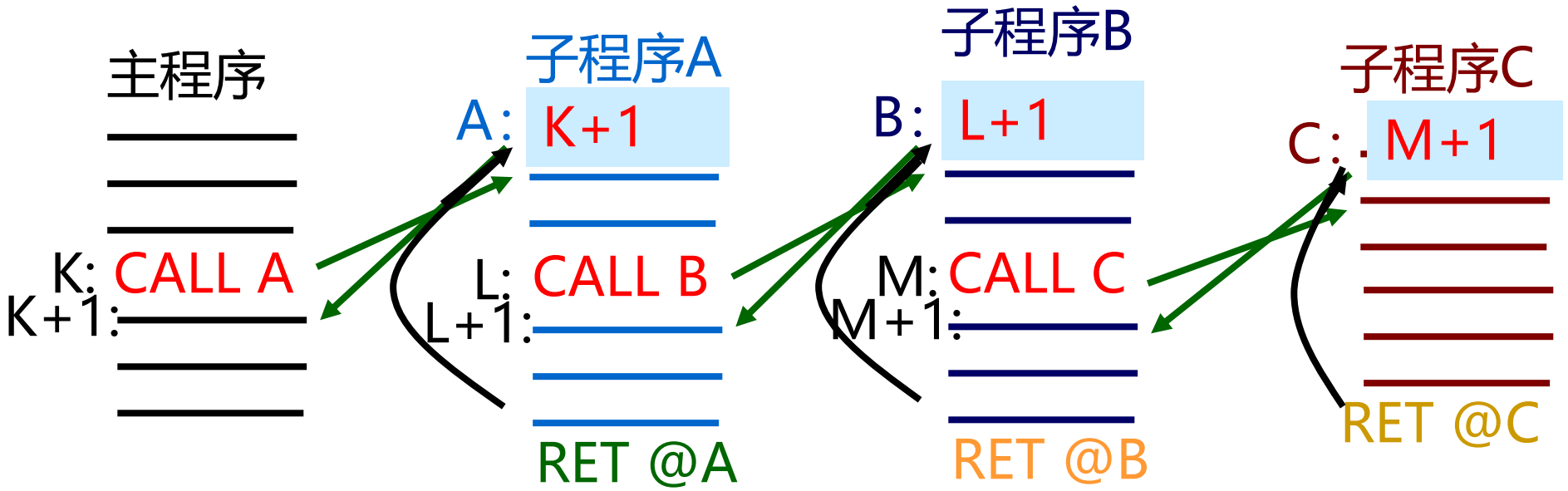
保存返回地址的方法除去堆栈以外还有：

(1)用子程序的第一个字单元存放返回地址。转子指令把返回地址存放在子程序的第一个字单元中，子程序从第二个字单元开始执行。返回时将第一个字单元地址作为间接地址，采用间址方式返回主程序。这种方法可以实现多重转子，但不能实现递归循环，Cyber70采用的就是这种方法。

(2)用寄存器存放返回地址。转子指令先把返回地址放到某一个寄存器中，再由子程序将寄存器中的内容转移到另一个安全的地方，比如存储器的某个区域。这是一种较为安全的方法，可以实现子程序的递归循环。IBM370采用这种方法，这种方法相对增加了子程序的复杂程度。

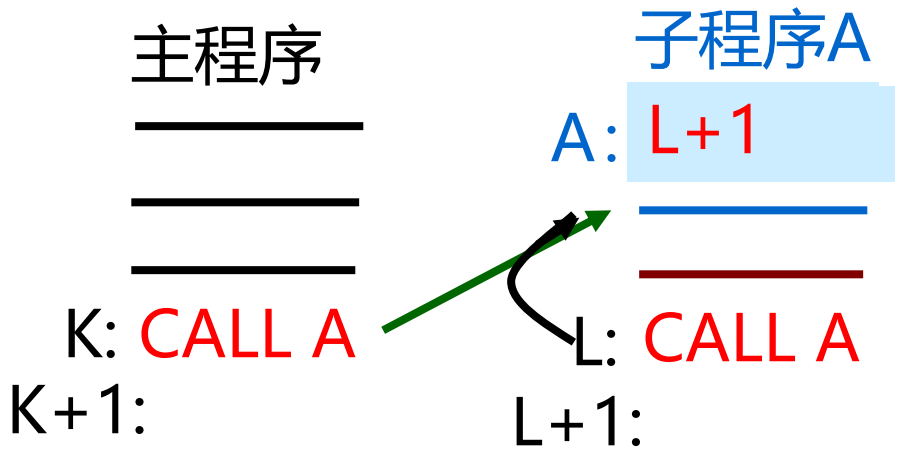
# 子程序嵌套

(利用子程序的第一个单元保存返回地址)



# 子程序递归

(利用子程序的第一个单元保存返回地址)



## 3.返回指令

从子程序转向主程序的指令称为返回指令，其助记符一般为RET，子程序的最后一条指令一定是返回指令。

返回地址存放的位置决定了返回指令的格式，如果返回地址保存在堆栈中，则返回指令是零地址指令，如果返回地址保存在某个主存单元中，则返回指令中必须是一地址指令。

# 输入/输出类指令



.4 输入/输出（I/O）类指令用来实现主机与外部设备之间的信息交换，包括输入/输出数据、主机向外设发控制命令或外设向主机报告工作状态等。从广义的角度看，I/O指令可以归入数据传送类。

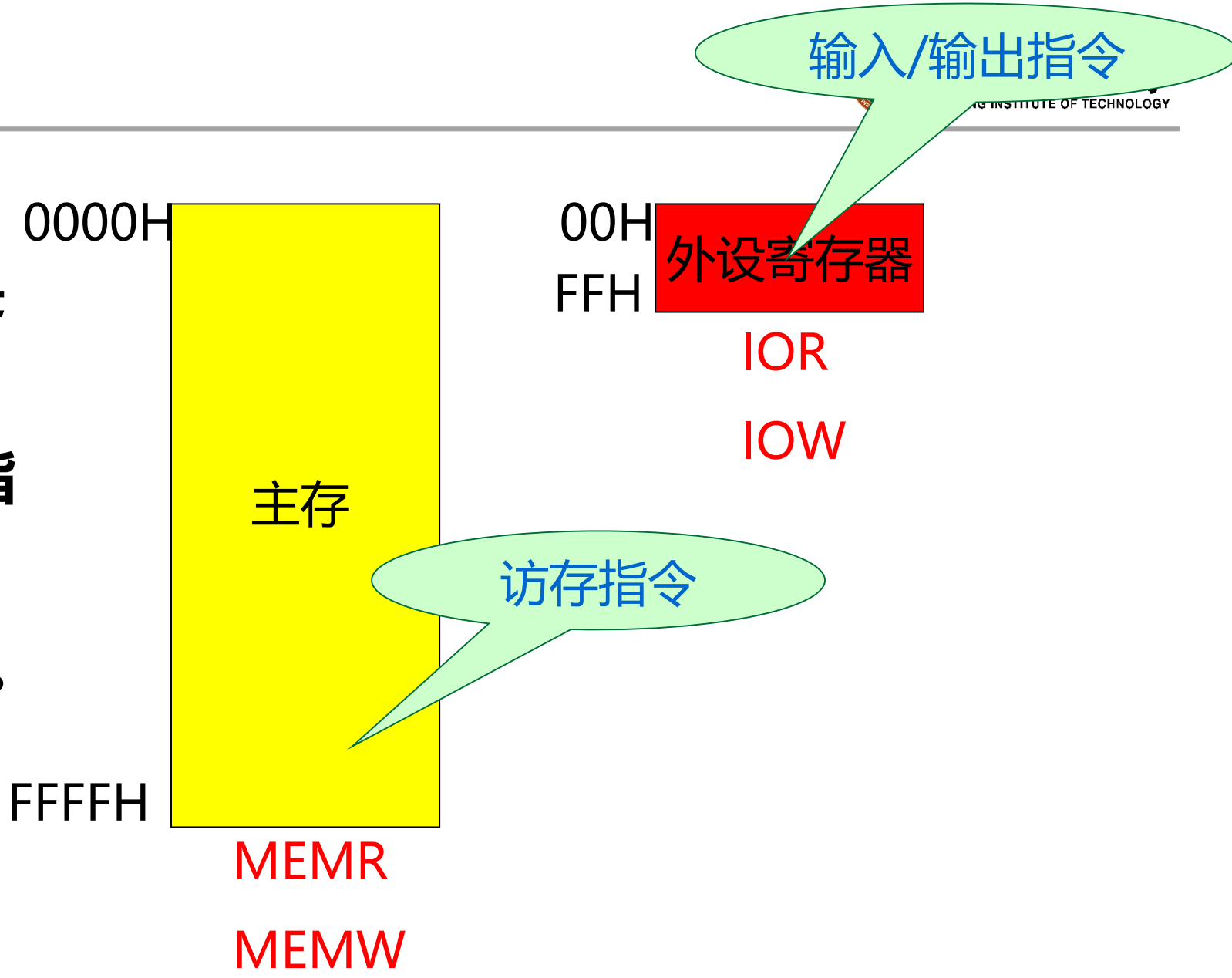
各种不同的计算机的I/O指令差别很大，通常有两种方式：

独立编址方式

统一编址方式

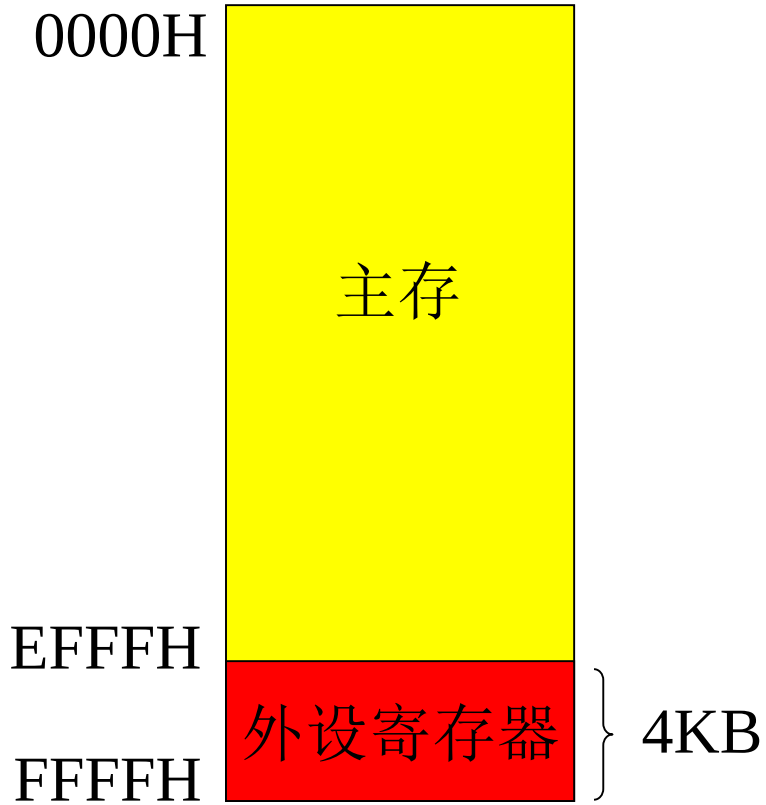
## 1.独立编址的I/O

所谓独立编址就是把外设端口和主存单元分别独立编址。指令系统中有专门的**IN/OUT**指令。以主机为基准，信息由外设传送到主机称为输入，反之称为输出。指令中需要给出外设端口地址。这些端口地址与主存地址无关，是另一个独立的地址空间。



## 2.统一编址的I/O

所谓统一编址就是把外设寄存器和主存单元统一编址。指令系统中没有专门的I/O指令，就用一般的数据传送类指令来实现I/O操作。




数据传送指令

输入指令

MOV R$_0$,(30H)

MOV R$_0$,(F000H)

## 复杂指令系统计算机

### CISC (Complex Instruction Set Computer)

早期计算机部件昂贵、速度慢，为了扩展硬件功能，不得不将更多更复杂指令加入到指令系统，以提高计算机的处理能力，如早期x86架构的指令集。

### 主要特点

①指令数量多；

②指令长度可以不固定，指令格式和寻址方式多样；

③很多指令会涉及存储器读写操作，指令周期长；

④多采用微程序控制方式。

## CISC指令集面临的主要问题

日趋庞大的指令系统，控制逻辑不规整，**设计与实现困难，调试和维护难度增加**，而且因指令操作复杂而增加机器周期，从而**降低了系统性能**。

## CISC指令集的“20%-80%律”。

在程序中各种指令出现的频率悬殊很大，占指令总数20%的简单指令在程序中的出现频率占80%，而占指令总数80%的复杂指令在程序中的出现频率只占约20%。

## 精简指令系统计算机

RISC (Reduce Instruction Set Computer)

现代新的指令集大多采用RISC体系结构，如ARM、MIPS、RISC(-I到-V）指令集。X86发展过程中也借鉴了RISC思想。

## 主要特点

①指令数量少；

②指令长度固定，指令格式和寻址方式种类少；

③大部分采用RR寻址，采用Load/Store指令读写存储器，指令周期短；

④采用组合逻辑电路控制，不用或少用微程序控制。

## VLIW和EPIC

VLIW是英文“Very Long Instruction Word”的缩写，中文含义是“超长指令字”，即一种非常长的指令组合，它把许多条指令连在一起，增加了运算的速度。在这种指令系统中，**编译器**把许多简单、独立的指令组合到一条指令字中。当这些指令字从主存中取出放到处理器中时，它们被容易地分解成几条简单的指令，这些简单的指令被分派到一些独立的执行单元去执行。

VLIW提供了一条把CISC指令分解成基本RISC指令的方法。

超长指令字(VLIW)是指令级并行，超线程(HT)是线程级并行，而多内核则是芯片级并行。这三种方式都是提高并行计算性能的有效途径。

EPIC是英文“Explicit Parallel Instruction Code”的缩写，中文含义是“显式并行指令代码”。 EPIC是从VLIW中衍生出来的，通过将多条指令放入一个指令字，有效的提高了CPU各个计算功能部件的利用效率，提高了程序的性能。

EPIC模式专为高效地并行处理而设计，能够同时处理多个指令或程序。

# 感谢聆听