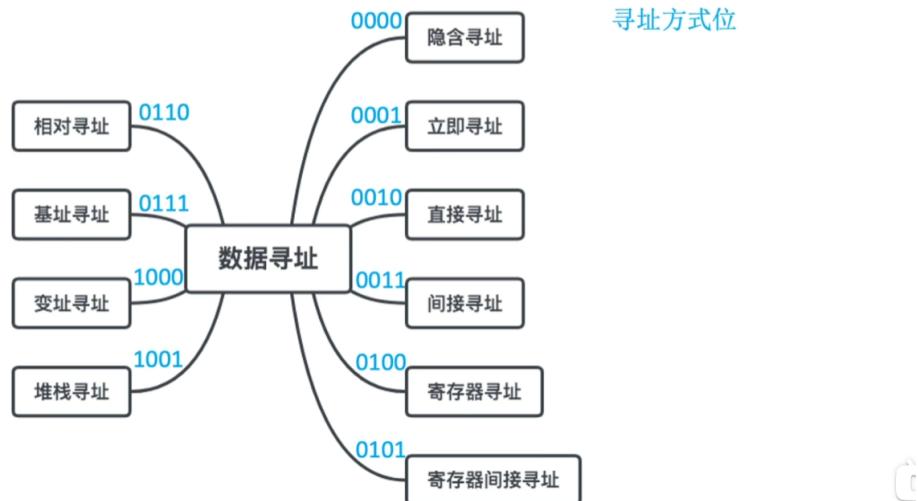


常见的数据寻址方式

寻址方式	有效地址	访存次数(指令执行期间)
隐含寻址	程序指定	0
立即寻址	A即是操作数	0
直接寻址	$EA=A$	1
一次间接寻址 Ø	$EA=(A)$	2
寄存器寻址	$EA=R_i$	0
寄存器间接一次寻址	$EA=(R_i)$	1
偏移寻址 转移指令 多道程序 循环程序	$EA=(PC)+A$ $EA=(BR)+A$ $EA=(IX)+A$	1 1 1
堆栈寻址	入栈/出栈时EA的确定方式不同	硬堆栈不访存，软堆栈访存1次

一地址指令 操作码 (OP) 寻址特征 形式地址 (A) 求出操作数的真实地址，称为有效地址(EA)。



二地址指令 操作码 (OP) 寻址特征 形式地址 (A_1) 寻址特征 形式地址 (A_2)

假设指令字长=机器字长=存储字长

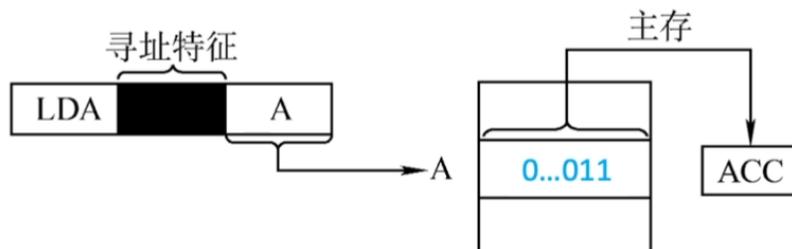
假设操作数为3

直接寻址

假设指令字长=机器字长=存储字长，操作数为3



直接寻址：指令字中的形式地址A就是操作数的真实地址EA，即 $EA=A$ 。



一条指令的执行：
取指令 访存1次
执行指令 访存1次
暂不考虑存结果
共访存2次

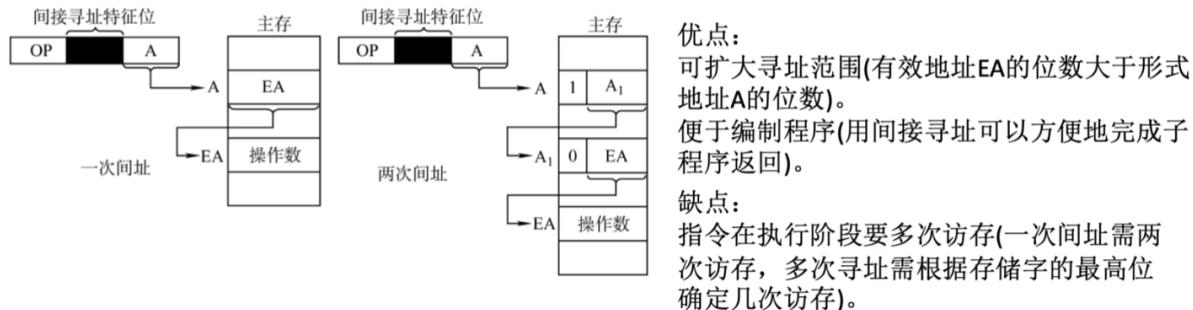
优点：简单，指令执行阶段仅访问一次主存，
不需专门计算操作数的地址。
缺点：
A的位数决定了该指令操作数的寻址范围。
操作数的地址不易修改。

间接寻址方式

假设指令字长=机器字长=存储字长，操作数为3



间接寻址：指令的地址字段给出的形式地址不是操作数的真正地址，而是操作数有效地址所在的存储单元的地址，也就是操作数地址的地址，即 $EA=(A)$ 。



寄存器寻址

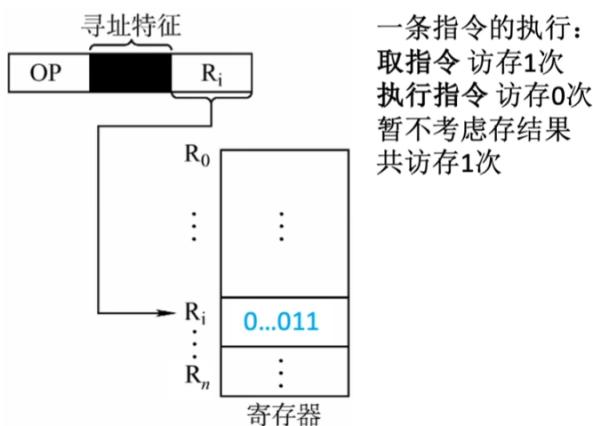
假设指令字长=机器字长=存储字长，操作数为3

一地址指令

操作码 (OP)

1001

寄存器寻址：在指令字中直接给出操作数所在的寄存器编号，即 $EA = R_i$ ，其操作数在由 R_i 所指的寄存器内。



一条指令的执行：
取指令 访存1次
执行指令 访存0次
暂不考虑存结果
共访存1次

优点：

指令在执行阶段不访问主存，只访问寄存器，
指令字短且执行速度快，支持向量/矩阵运算。

缺点：

寄存器价格昂贵，计算机中寄存器个数有限。

寄存器间接寻址

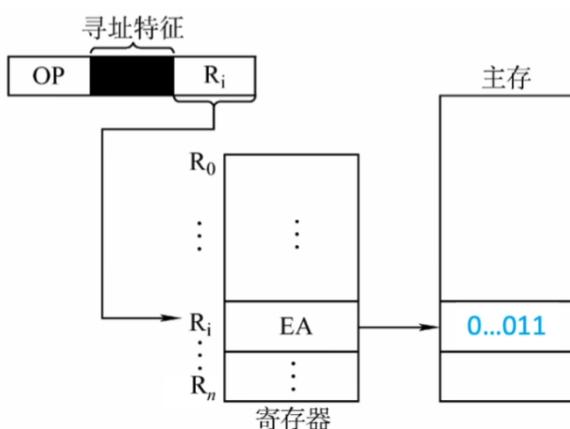
假设指令字长=机器字长=存储字长，操作数为3

一地址指令

操作码 (OP)

1001

寄存器间接寻址：寄存器 R_i 中给出的不是一个操作数，而是操作数所在主存单元的地址，
即 $EA = (R_i)$ 。



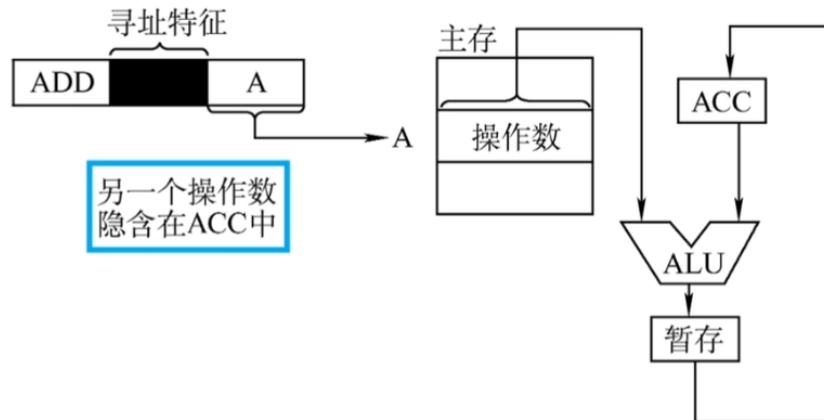
一条指令的执行：
取指令 访存1次
执行指令 访存1次
暂不考虑存结果
共访存2次

特点：

与一般间接寻址相比速度更快，但指令的执行阶段需要访问主存(因为操作数在主存中)。

隐含寻址

隐含寻址：不是明显地给出操作数的地址，而是在指令中隐含着操作数的地址。



优点：有利于缩短指令字长。

缺点：需增加存储操作数或隐含地址的硬件。

立即寻址

假设指令字长=机器字长=存储字长，操作数为3

一地址指令	操作码 (OP)	#	0...011
-------	----------	---	---------

立即寻址：形式地址A就是操作数本身，又称为立即数，一般采用补码形式。

#表示立即寻址特征。

一条指令的执行：

优点：指令执行阶段不访问主存，指令执行时间最短

取指令 访存1次

①

缺点：

执行指令 访存0次

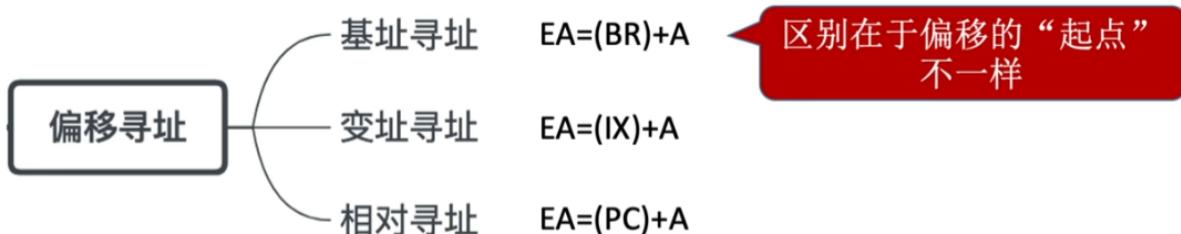
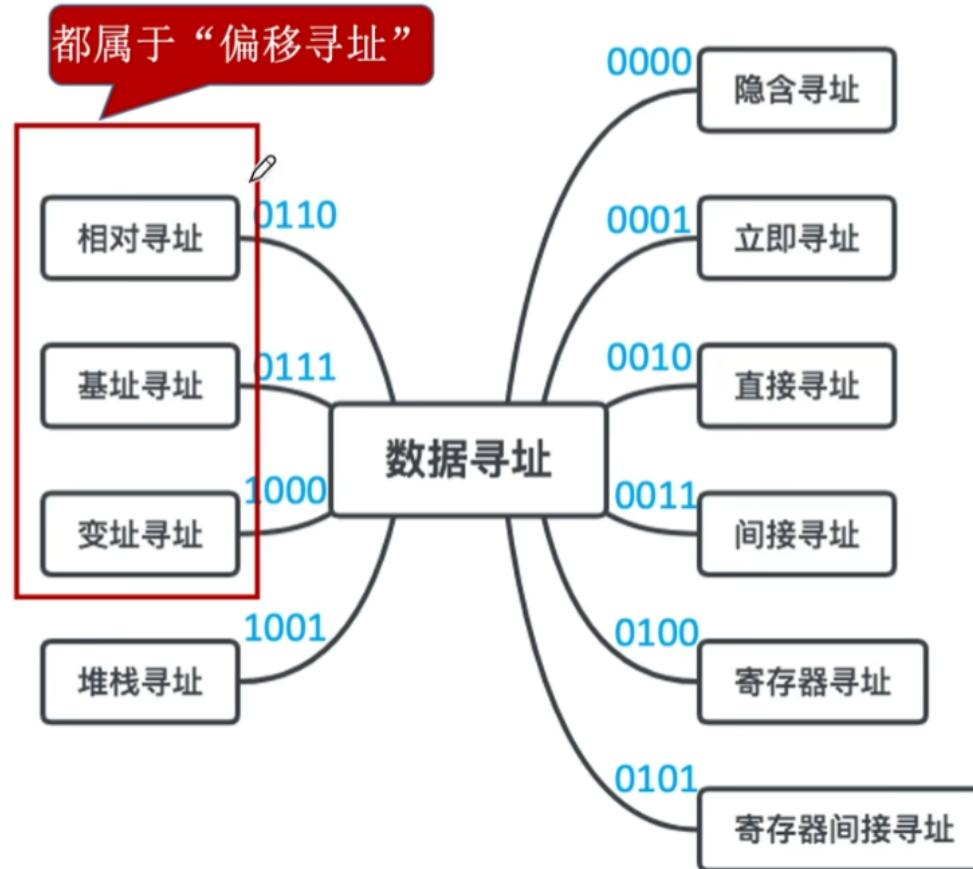
A的位数限制了立即数的范围。

暂不考虑存结果

如A的位数为n，且立即数采用补码时，可表示的数据范围为 $-2^{n-1} \sim 2^{n-1} - 1$

共访存1次

偏移寻址



基址寻址：以程序的起始存放地址作为“起点”

变址寻址：程序员自己决定从哪里作为“起点”

相对寻址：以程序计数器PC所指地址作为“起点”

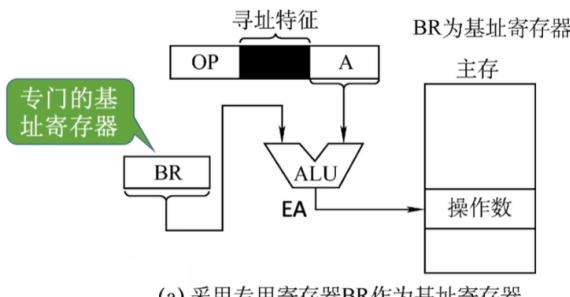
基址寻址

基址寻址

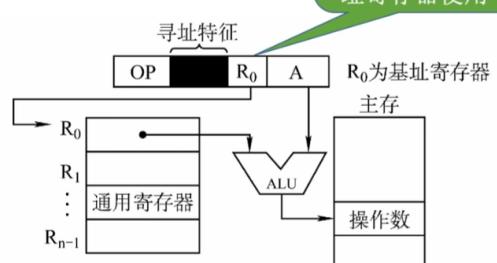
注: BR—base address register
EA—effective address

基址寻址: 将CPU中基址寄存器(BR)的内容加上指令格式中的形式地址A, 而形成操作数的有效地址, 即 $EA=(BR)+A$ 。

在指令中指明,
要将哪个通用
寄存器作为基
址寄存器使用



(a) 采用专用寄存器BR作为基址寄存器



(b) 采用通用寄存器作为基址寄存器

Tips: 可对比操作系统第三章第一节学习,
OS课中的“重定位寄存器”就是“基址寄存器”

要用几个bit
指明寄存器?

根据通用寄存器总数判断



稍加思考

基址寻址的作用

拓展: 程序运行前, CPU将BR的值修改为该程序的起始地址(存在操作系统PCB中)

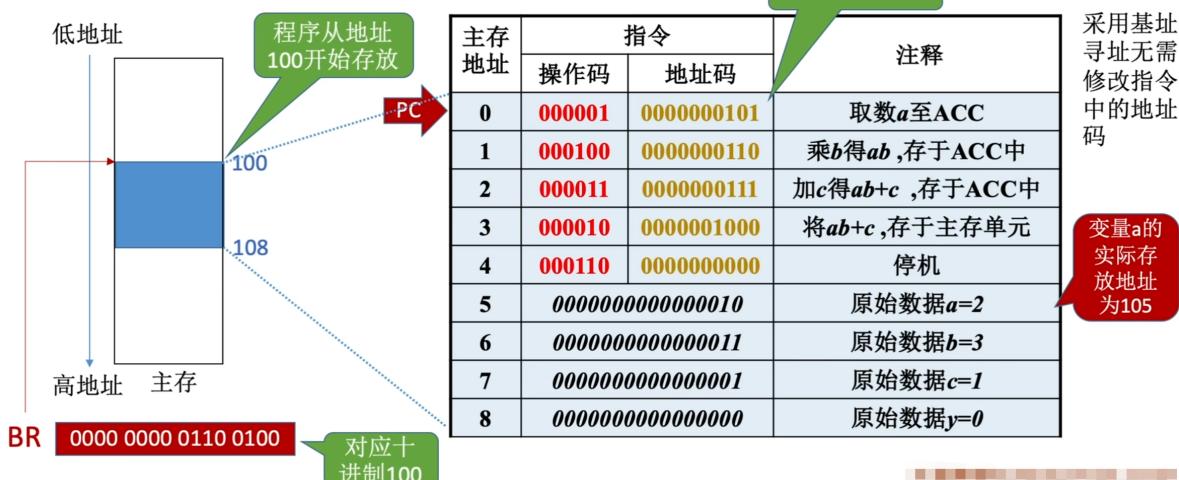
基址寻址的作用

```
int a=2,b=3,c=1,y=0;  
void main(){  
    y=a*b+c;  
}
```

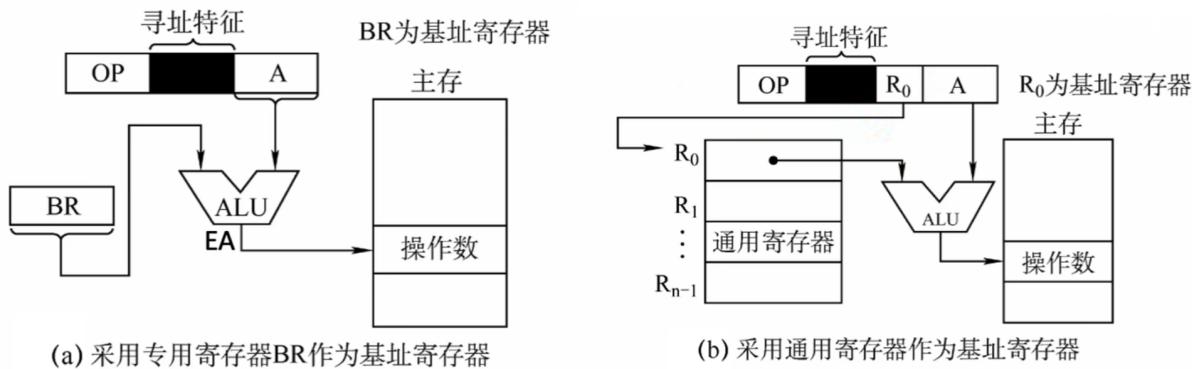
基址寻址: 将CPU中基址寄存器(BR)的内容加上指令格式中的形式地址A, 而形成操作数的有效地址, 即 $EA=(BR)+A$ 。

优点: 便于程序“浮动”, 方便实现多道程序并发运行

形式地址A = 5



基址寻址：将CPU中**基址寄存器（BR）**的内容加上指令格式中的形式地址A，而形成操作数的有效地址，即 **$EA = (BR) + A$** 。



注：基址寄存器是**面向操作系统的**，其**内容由操作系统或管理程序确定**。在程序执行过程中，基址寄存器的内容不变（作为基址），形式地址可变（作为偏移量）。

当采用通用寄存器作为基址寄存器时，可由**用户决定哪个寄存器作为基址寄存器**，但其**内容仍由操作系统确定**。

优点：可扩大寻址范围（基址寄存器的位数大于形式地址A的位数）；用户不必考虑自己的程序存于主存的哪一空间区域，故**有利于多道程序设计**，以及可用于**编制浮动程序**（整个程序在内存里边的浮动）。

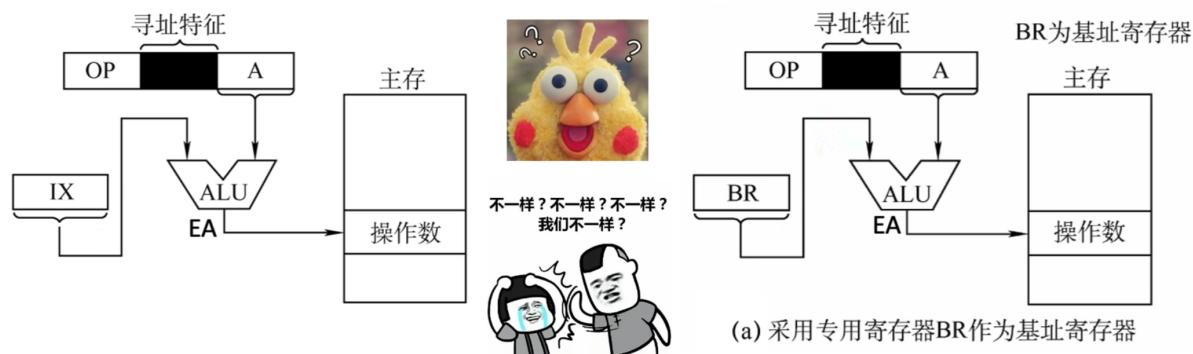
变址寻址

哔哩哔哩

变址寻址

注：IX — [index register](#)

变址寻址：有效地址EA等于指令字中的形式地址A与**变址寄存器IX**的内容相加之和，即 **$EA = (IX) + A$** ，其中**IX**可为变址寄存器（专用），也可用通用寄存器作为变址寄存器。



注：变址寄存器是**面向用户的**，在程序执行过程中，**变址寄存器的内容可由用户改变**（IX作为偏移量），**形式地址A不变**（作为基址）。

基址寻址中，BR保持不变作为基址，A作为偏移量

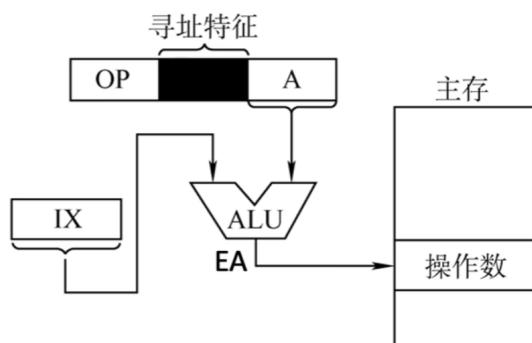
变址寻址的作用

变址寻址的作用

注：此处未添加“寻址特征”位，但实际上每条指令都会指明寻址方式。
此处讲解仅用口头描述



变址寻址：有效地址EA等于指令字中的形式地址A与变址寄存器IX的内容相加之和，即 $EA = (IX) + A$ ，其中IX可为变址寄存器（专用），也可用通用寄存器作为变址寄存器。



注：变址寄存器是面向用户的，在程序执行过程中，变址寄存器的内容可由用户改变（作为偏移量），形式地址A不变（作为基地址）。

优点：在数组处理过程中，可设定A为数组的首地址，不断改变变址寄存器IX的内容，可很容易形成数组中任一数据的地址，特别适合编制循环程序。

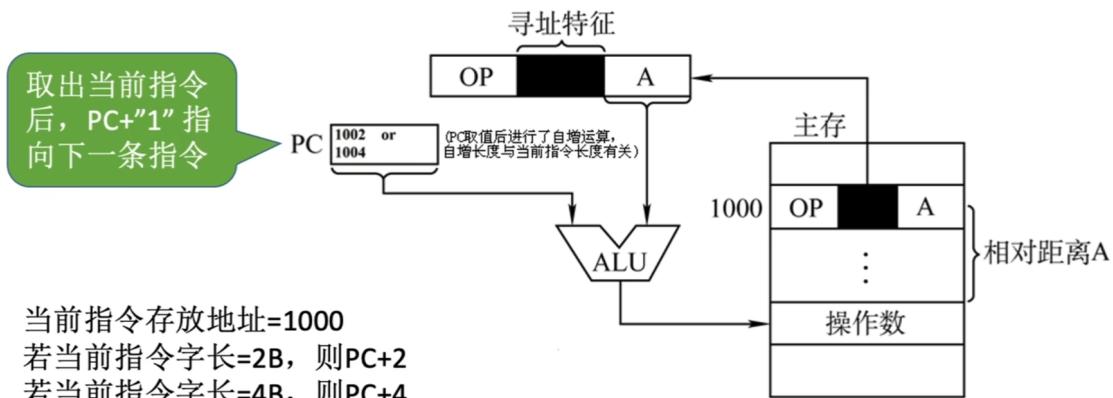
基址&变址复合寻址



相对寻址

相对寻址: 把程序计数器PC的内容加上指令格式中的形式地址A而形成操作数的有效地址, 即 $EA=(PC)+A$, 其中A是相对于PC所指地址的位移量, 可正可负, 补码表示。

注: 王道书的小错误——“A是相对于当前指令地址的位移量” ✗



因此取出当前指令后PC可能为 1002 or 1004

相对寻址的作用

拓展：ACC加法指令的地址码，可采用“分段”方式解决，即程序段、数据段分开。

```
for(int i=0; i<10; i++){
    sum += a[i];
}
```

问题：随着代码越写越多，你想挪动for循环的位置

相对寻址： $EA=(PC)+A$ ，其中A是相对于PC所指地址的位移量，可正可负，补码表示

优点：这段代码在程序内浮动时不用更改跳转指令的地
址码

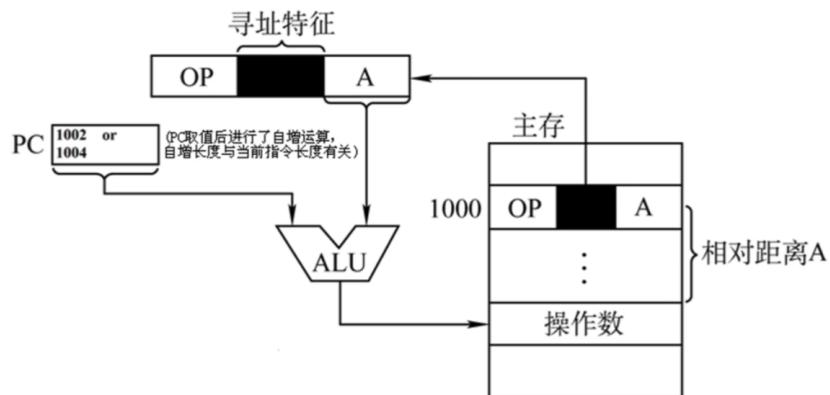
PC M+4

用相对寻址

相对寻址的作用

主存地址	指令		注释
	操作码	地址码	
0	取数到ACC	#0	立即数 0 → ACC
1	取数到IX	#0	立即数 0 → IX
2	其他代码
3	其他代码
4	其他代码
5	其他代码
...	其他代码
M	ACC加法	7 (数组始址)	(ACC)+(7+(IX)) → ACC
M+1	IX加法	#1	(IX) + 1 → IX
M+2	IX比较	#10	比较10-(IX)
M+3	条件跳转	-4(补码表示)	若结果>0 则PC跳转到M
M+4
...

相对寻址：把程序计数器PC的内容加上指令格式中的形式地址A而形成操作数的有效地址，即 $EA=(PC)+A$ ，其中A是相对于PC所指地址的位移量，可正可负，补码表示。



优点：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，因此便于程序浮动（一段代码在程序内部的浮动）。
相对寻址广泛应用于转移指令。

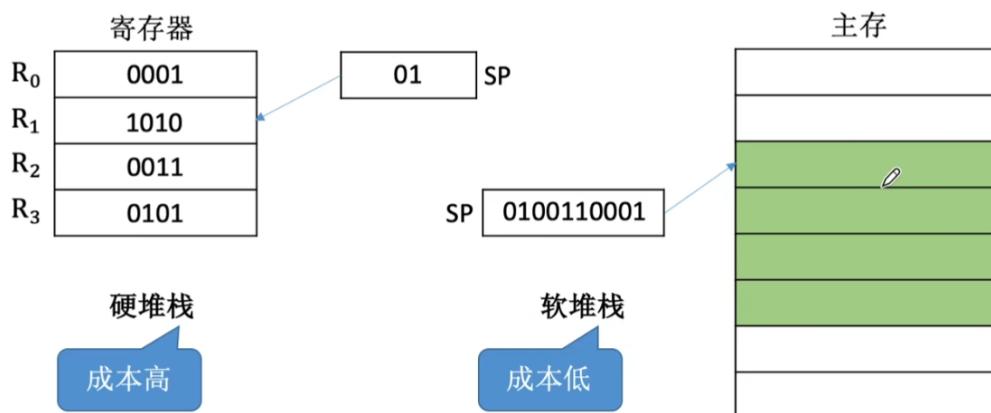
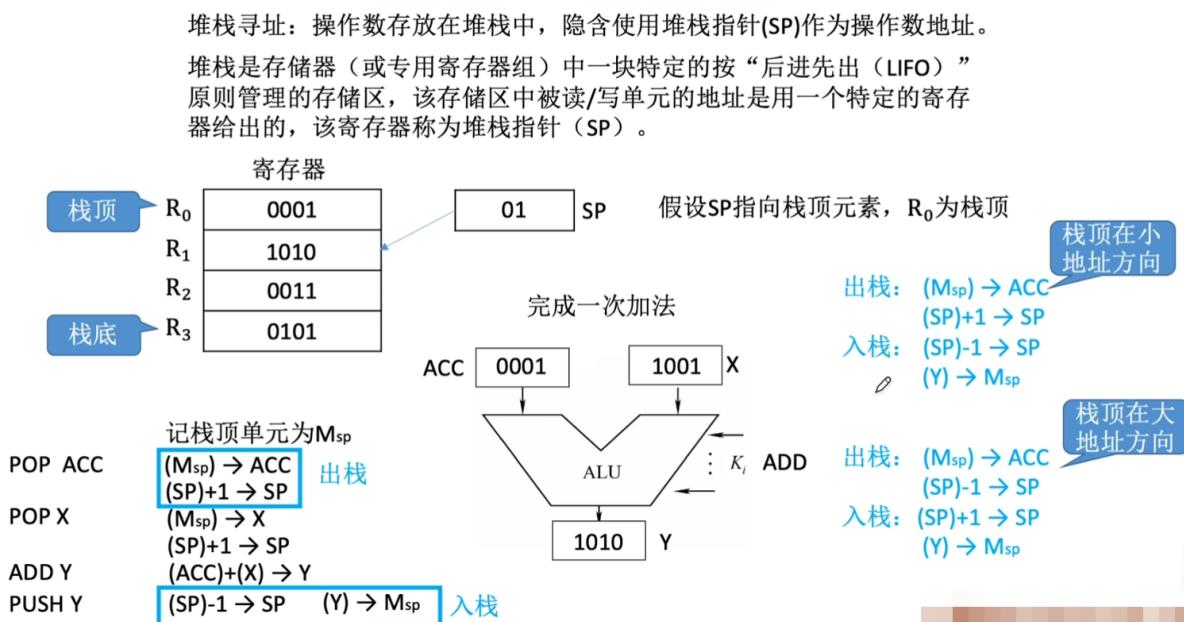
堆栈寻址

堆栈寻址：操作数存放在堆栈中，隐含使用堆栈指针(SP)作为操作数地址。

注：SP - Stack Pointer

堆栈寻址

注: SP — Stack Pointer



堆栈可用于函数调用时保存当前函数的相关信息(可参考数据结构“算法空间复杂度”的视频)

实现数的比较

高级语言视角:

```
if (a>b){  
} else {  
    ...  
    汇编语言中, 条件跳转指令有很多种, 如 je 2 表示当比较结果为 a=b 时跳转到2  
    ...  
    jg 2 表示当比较结果为 a>b 时跳转到2  
}
```

硬件视角:

- 通过“**cmp**指令”比较 a 和 b (如 **cmp a, b**) , 实质上是用 a-b
- 相减的结果信息会记录在程序状态字寄存器中 (PSW)
- 根据PSW的某几个标志位进行条件判断, 来决定是否转移

- PSW中有几个比特位记录上次运算的结果
- 进位/借位标志 CF: 最高位有进位/借位时 CF=1
 - 零标志 ZF: 运算结果为0则 ZF=1, 否则 ZF=0
 - 符号标志 SF: 运算结果为负, SF=1, 否则为0
 - 溢出标志 OF: 运算结果有溢出 OF=1否则为0

硬件如何实现数的“比较”

注: 无条件转移指令 jmp 2, 就不会管PSW的各种标志位

主存地址	指令		注释
	操作码	地址码	
0	取数到ACC	#0	立即数 0 → ACC
1	取数到IX	#0	立即数 0 → IX
2	ACC加法	7 (数组始址)	(ACC)+(7+(IX))→ ACC
3	IX加法	#1	(IX)+1 → IX
4	IX比较	#10	比较10-(IX)
5	条件跳转	2	若结果>0 则PC跳转到2
6	从ACC存数	17	(ACC)→ sum变量
7	随便什么值		a[0]
8	随便什么值		a[1]
9	随便什么值		a[2]
...
16	随便什么值		a[9]
17	初始为0		sum变量