计算机组织结构

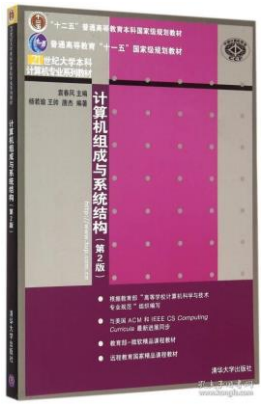
14 指令系统

刘博涵

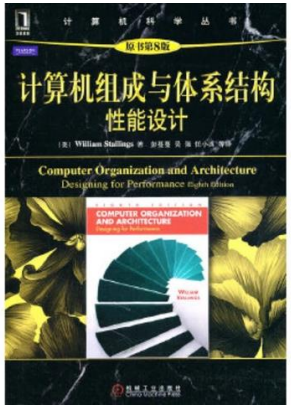
2022年12月1日

南京大学
NANJING UNIVERSITY

教材对应章节



第4章 指令系统



第10章 指令集：特征和功能
第11章 指令集：寻址方式和指令格式

计算机功能

- 计算机的基本功能是执行程序
- 程序由存储在内存中的一组指令组成
- CPU通过执行指定的指令来完成实际工作

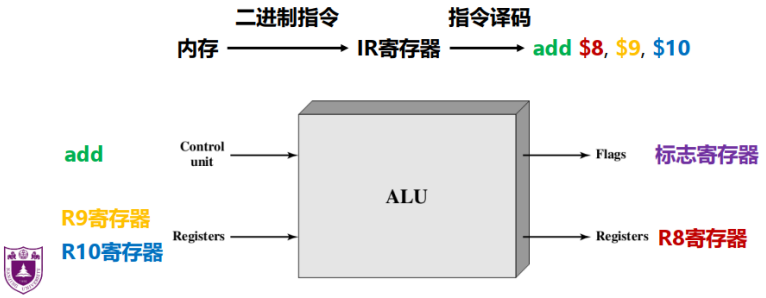
- 指令集：CPU能执行的各种不同指令的集合

回顾：指令

- 指令是计算机处理的最基本单位
  - 操作码（指令执行的内容）+ 操作数（要操作的对象）
- 多周期实现方案
  - 可以将一条指令的执行分解为一系列步骤
    - 取指令，译码/取寄存器，执行/有效地址/完成分支，访问内存，存储结果

算术逻辑单元 (ALU)

- 算术逻辑单元 (ALU) 是计算机实际完成数据算术逻辑运算的部件
  - 数据由寄存器 (Registers) 提交给ALU，运算结果也存于寄存器
  - ALU可能根据运算结果设置一些标志 (Flags)，标志值也保存在处理器内的寄存器中
  - 控制器 (Control unit) 提供控制ALU操作和数据传入送出ALU的信号

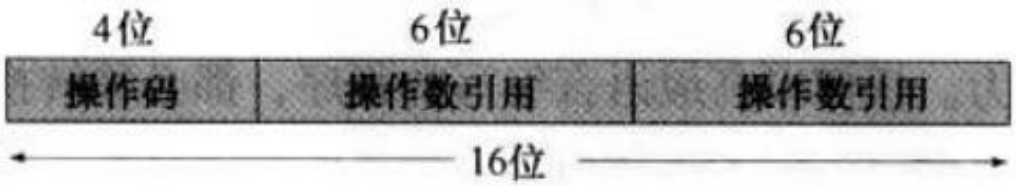


指令的要素

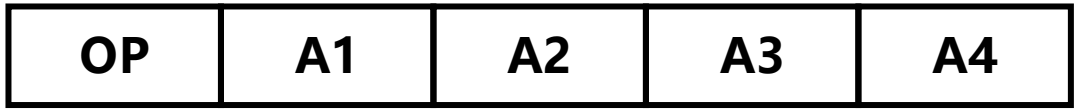
- **操作码**：指定将要完成的操作
- **源操作数引用**：操作会涉及一个或多个源操作数，这是操作所需的输入
- **结果操作数引用**：操作可能会产生一个结果
- **下一指令引用**：告诉处理器这条指令执行完成后到哪儿去取下一条指令

指令表示

- 在计算机内部，指令由一个位串来表示
- 指令格式：对应于指令的各要素，这个位串划分成几个字段
  - 大多数指令集使用不止一种指令格式　如何避免指令的二义性?
- 机器指令符号表示法：
  - 操作码被缩写成助记符来表示
    - ADD：加，SUB：减，MUL：乘，DIV：除，LOAD：由存储器装入，STOR：保存到存储器…
  - 操作数也可以用符号表示
    - 用寄存器编号或内存地址替换操作数

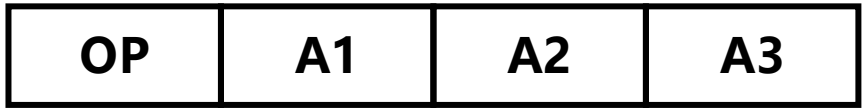


指令格式



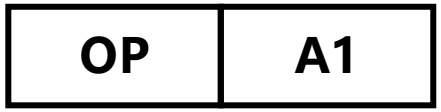
(A1) OP (A2) -> A3

A4为下一条将要执行指令的地址



(A1) OP (A2) -> A3

(A1) OP (A2) -> A1



1）OP (A1) -> A1

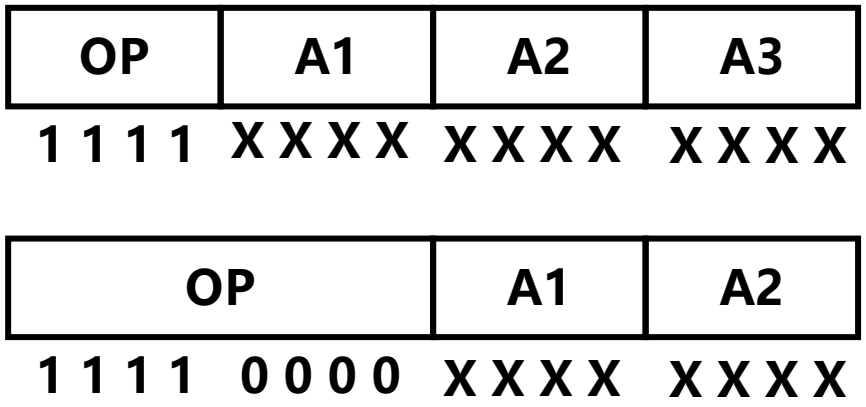
2）(ACC累加器) OP (A1) -> ACC

1）不需要操作数的指令：如空操作指令、停机指令、关中断指令

2）堆栈计算机中的应用：操作数从栈中弹出

指令格式：扩展操作码

- 定长操作码
  - n位操作码字段的指令系统最大能够表示2^n条指令
- 可变长操作码
  - **扩展操作码**：不同地址数的指令具有不同长度的操作码



怎么区分是一个三地址指令还是一个二地址指令?

15条三地址指令操作码：0 0 0 0 ~ 1 1 1 0
15条二地址指令操作码：1 1 1 1 0 0 0 0 ~ 1 1 1 1 1 1 1 0
15条一地址指令操作码：1 1 1 1 1 1 1 1 0 0 0 0 ~ 1 1 1 1 1 1 1 1 1 1 1 0
16条零地址指令操作码：剩余的

指令格式

- 指令格式通过它的各个构成部分来定义指令的位安排
- 一个指令格式必须包含一个操作码，以及隐式或显式的、零个或多个操作数
- 指令格式必须显式或隐式地为每个操作数指定其寻址方式
- **大多数指令集使用不止一种指令格式**

指令格式：例题

- **例题：**假设指令字长为16位，操作数的地址码为6位，指令有零地址、一地址、二地址3种格式。
  - 设操作码固定，若零地址指令有M种，一地址指令有N种，则二地址指令最多有几种？

    操作码位数：16 – 6 – 6 = 4
    总指令条数：2^4 = 16
    二地址指令最多：16 – M – N
  - 采用扩展操作码技术，二地址指令最多有几种？

    0000 ~ 1110
  - 采用扩展操作码技术，若二地址指令有P条，零地址指令有Q条，则一地址指令最多有几种？

    $Q = [(2^4 - P) \times 2^6 - R] \times 2^6$

操作码

- **差异**：不同的计算机上操作码的数目变动是很大的
- **共性**：所有计算机上都会存在相同的常用操作类型
  - 数据传送
  - 算术运算
  - 逻辑运算
  - 转换
  - 输入/输出
  - 系统控制
  - 控制转移

操作码：数据传送

- 指明源和目标操作数的位置
- 指明将要传送数据的长度
- 指明每个操作数的寻址方式

数据传送	Move(transfer)	由源向目标传送字或块
	Store	由处理器向存储器传送字
	Load(fetch)	由存储器向处理器传送字
	Exchange	源和目标交换内容
	Clear(reset)	传送全 0 字到目标
	Set	传送全 1 字到目标
	Push	由源向栈顶传送字
	Pop	由栈顶向目标传送字

操作码：算术运算

- 一条算术指令的执行会涉及数据传送操作，来为算术和逻辑单元准备输入，并传送逻辑单元的输出

算术运算	Add	计算两个操作数的和
	Subtract	计算两个操作数的差
	Multiply	计算两个操作数的积
	Divide	计算两个操作数量商
	Absolute	以其绝对值替代操作数
	Negate	改变操作数的符号
	Increment	操作数加 1
	Decrement	操作数减 1

操作码：逻辑运算

- 位操作：操作一个字或其它可寻址单元的中的个别位
- 移位和旋转

逻辑运算	AND	执行逻辑与操作
	OR	执行逻辑或操作
	NOT(complement)	执行逻辑非操作
	Exclusive-OR	执行逻辑异或操作
	Test	测试指定的条件；根据结果设置标志
	Compare	对两个或多个操作数进行逻辑的或算术的比较；根据结果设置标志
	Set 控制变量	出于保护目的、中断管理、时间控制等原因进行设置控制的一类指令
	Shift	左（右）移位操作数，一端引入常数
	Rotate	循环左（右）移位操作数

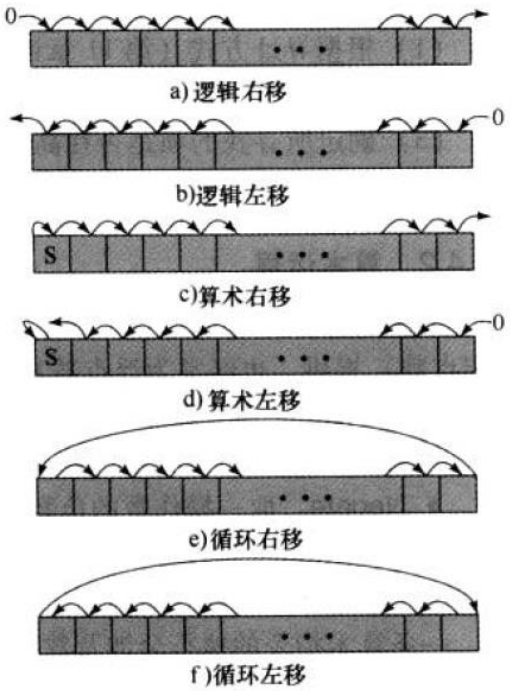


图 10-6　移位和旋转操作

操作码：输入/输出

- 各种输入/输出方法仅有少数输入/输出指令实现，具体操作由参数、代码或命令字指定

输入/输出（I/O）	Input(read)	由指定的 I/O 端口或设备传送数据到目标（如主存或处理器寄存器）
	Output(write)	由指定的源传送数据到 I/O 端口或设备
	Start I/O	向 I/O 处理器传送指令以初始化 I/O 操作
	Test I/O	由 I/O 系统向指定目标传送状态信息

操作码：控制转移

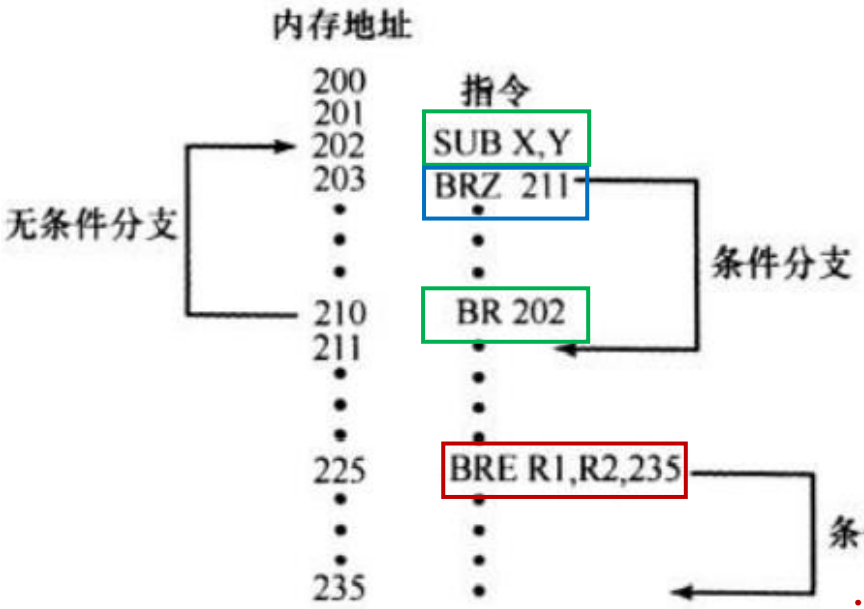
- 分支指令（亦称为跳转指令）
- 跳步指令
- 过程调用指令

控制转移	Jump(branch)	无条件转移；向 PC 装入指定地址
	Jump 条件	测试指定的条件，根据条件或将指定地址装入 PC 或什么也不做
	Jump 子程序	将当前程序控制信息放到一个已知位置；转移到指定地址
	Return	用已知位置内容替代 PC 和其他寄存器的内容
	Execute	由指定位置取操作数并作为指令执行；不修改 PC
	Skip	PC 加 1 以跳过下一指令
	Skip 条件	测试指定条件；基于条件或跳步或不跳步
	Halt	停止程序执行
	Wait(hold)	暂停程序执行；重复测试指定条件；当条件满足时恢复执行
	No operation	无操作完成；但程序执行继续

操作码：控制转移（续）

- **分支 / 跳转指令：**
  - 把将要执行的下一条指令的地址作为它的操作数之一

while (X-Y>0) {
    202~210
}



if (X-Y==0){
    211
}

if (R1的数据==R2的数据){
    235
}

操作码：控制转移（续）

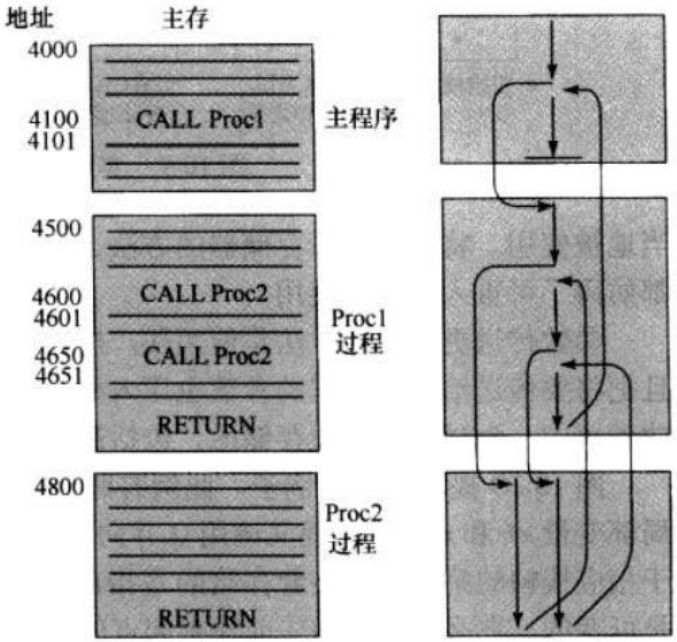
- **跳步指令：**
  - 包含一个隐含地址，该隐含地址等于下一指令地址加上该指令长度之和

```
301
•
•
•
309   ISZ   R1
310   BR    301
311
```

```
R1 = R1 + 1
if (R1==0){
   311 (即跳过了下一条指令310)
}
else {
   310
}
```

操作码：控制转移（续）

- **过程调用指令：**
  - 涉及由目前位置转移到过程的调用指令和由过程返回到调用发生位置的返回指令



使用寄存器

$RN \longleftarrow PC + \Delta$

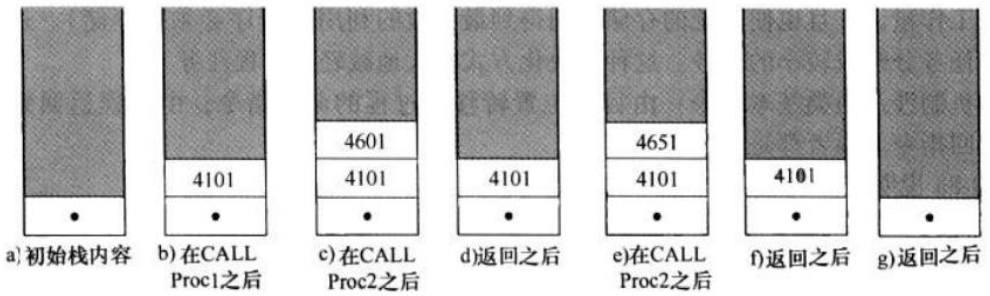
$PC \longleftarrow X$

返回地址存于过程开始处

$X \longleftarrow PC + \Delta$

$PC \longleftarrow X + 1$

使用栈



操作数

- 常见类型
  - 地址
  - 数值
  - 字符
  - 逻辑数据

操作数：地址

- 一个指令需要有4个地址引用：2个源操作数，1个目的操作数，以及下一指令地址
  - 下一指令地址可以是隐含的
  - 源操作数可以和目的操作数是相同的
- 地址数量
  - 每条指令中的地址数目越少
    - 指令的长度越短，不需要复杂的CPU
    - 使程序总的指令条数更多，导致执行时间更长，程序也更长更复杂
  - 对于多地址指令，普遍具有多个通用寄存器可用，允许某些运算只使用寄存器即可完成，从而使执行加快

指令		注释
SUB	Y, A, B	Y ← A − B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

指令		注释
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y − B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

指令		注释
LOAD	D	AC ← D
MPY	E	AC ← AC × E
ADD	C	AC ← AC + C
STOR	Y	Y ← AC
LOAD	A	AC ← A
SUB	B	AC ← AC − B
DIV	Y	AC ← AC ÷ Y
STOR	Y	Y ← AC

操作数：数值

- 计算机存储的数值是受限的
  - 机器可表示数值的幅值是有限的
  - 浮点数情况下数值精度是有限的
- 数值数据的类型
  - 二进制整数或定点数
  - 二进制浮点数
  - 十进制数

操作数：字符

- 国际参考字母表（IRA） / 美国信息交换标准码（ASCII）： 每个字符被表示成唯一的7位二进制串
- 扩展的二进制编码的十进制交换码（EBCDIC）： 8位编码
- 统一码（Unicode）： 16 位 / 32 位

操作数：逻辑数据

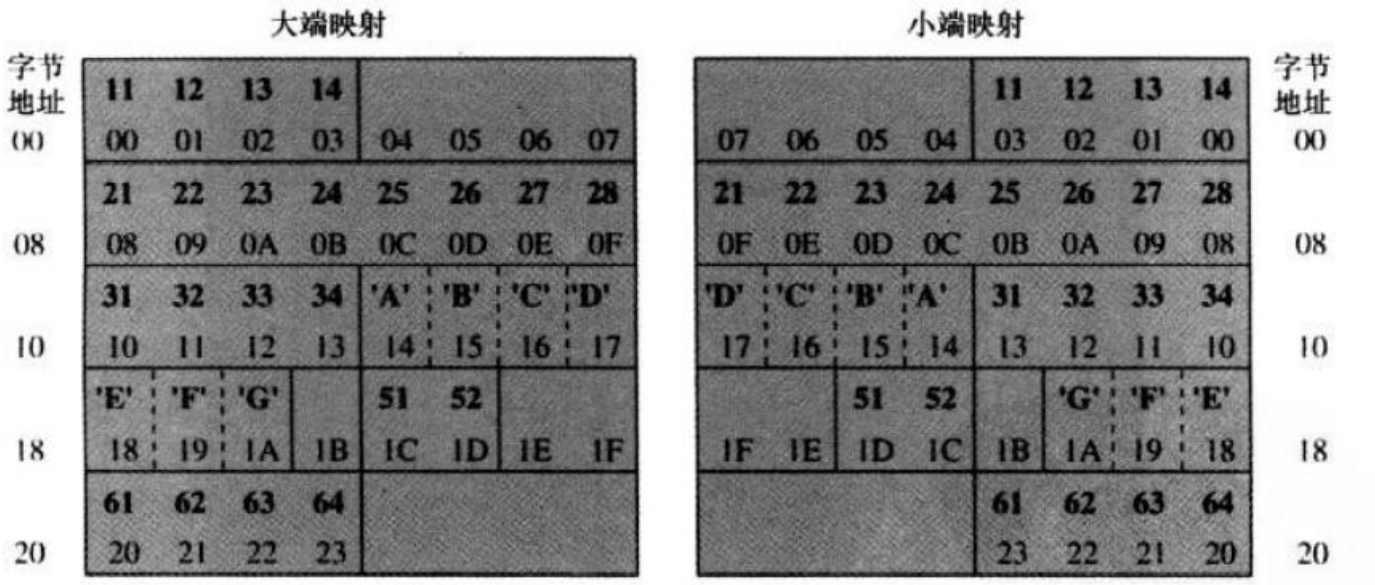
- 将一个n位单元看成是由n个1位项组成，每项有值0或1
  - 存储一个布尔或者二进制数据项序列，序列中的每个值只能取值1（真）或0（假）
  - 有利于实现对数据项的具体位进行操纵

操作数：大端序和小端序

- 假设有32位的十六进制值12345678，将它存储在可字节寻址的存储器地址184处

操作数：大端序和小端序（续）

- 在两种策略中每个数据项有同样地址（范围）
- 在任何一个给定的多字节值中，小端的字节排序是大端的反序，反之亦然
- 端序不影响结构中数据项的次序



操作数引用

- 操作数的实际值
- 操作数的地址
  - 寄存器
  - 主存 / 虚拟内存
  - ...

寻址方式

- 立即寻址
- 直接寻址
- 间接寻址
- 寄存器寻址
- 寄存器间接寻址
- 偏移寻址
- 栈寻址

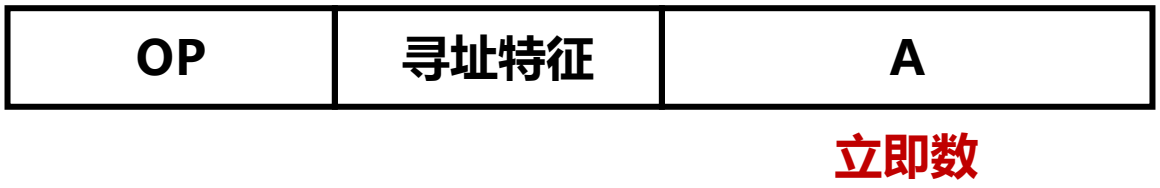
记号

- A: 指令中地址字段的内容
- R: 指向寄存器的指令地址字段内容
- EA: 被访问位置的实际（有效）地址
- (X): 存储器位置 X 或寄存器 X 的内容

操作数 = (EA)

立即寻址

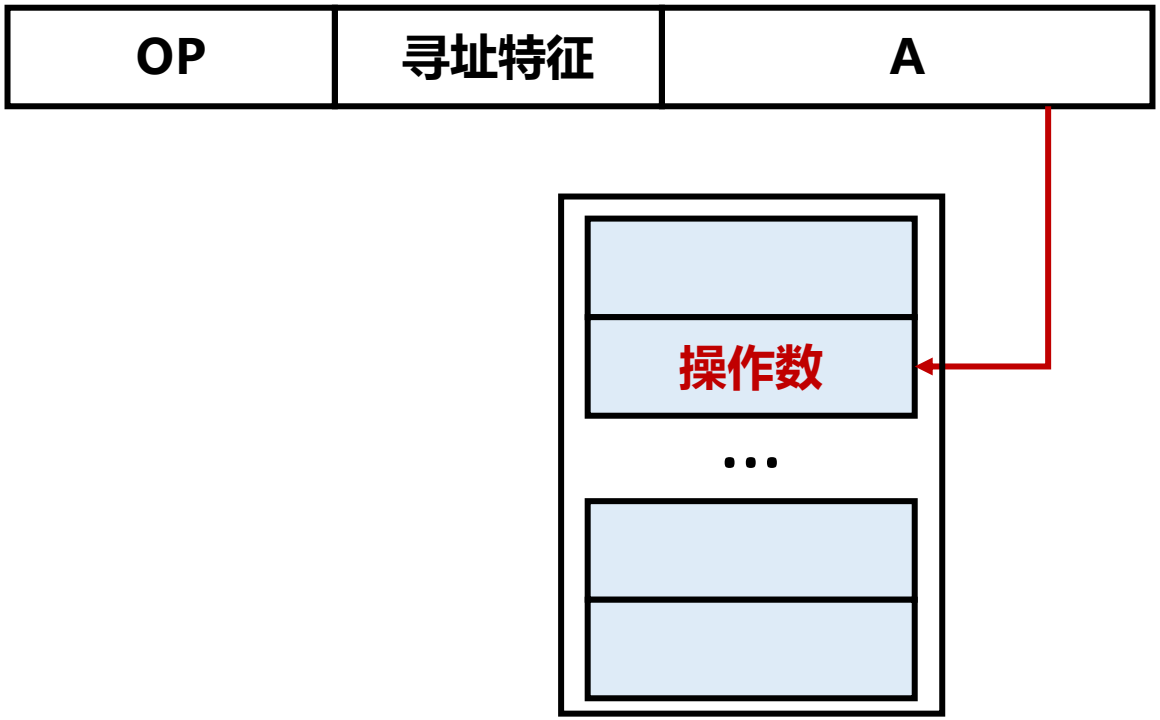
- **方式**：操作数实际值出现在指令中
- **用法**：定义和使用常数或设置变量的初始值
- **算法**：操作数 = A
- **优点**：快，除了取指令之外，获得操作数不要求另外的存储器访问
- **缺点**：数的大小受限于地址字段的长度



一个操作数**直接**放在指令里面，通过指令就能读到这个操作数，
是立即寻址还是直接寻址？

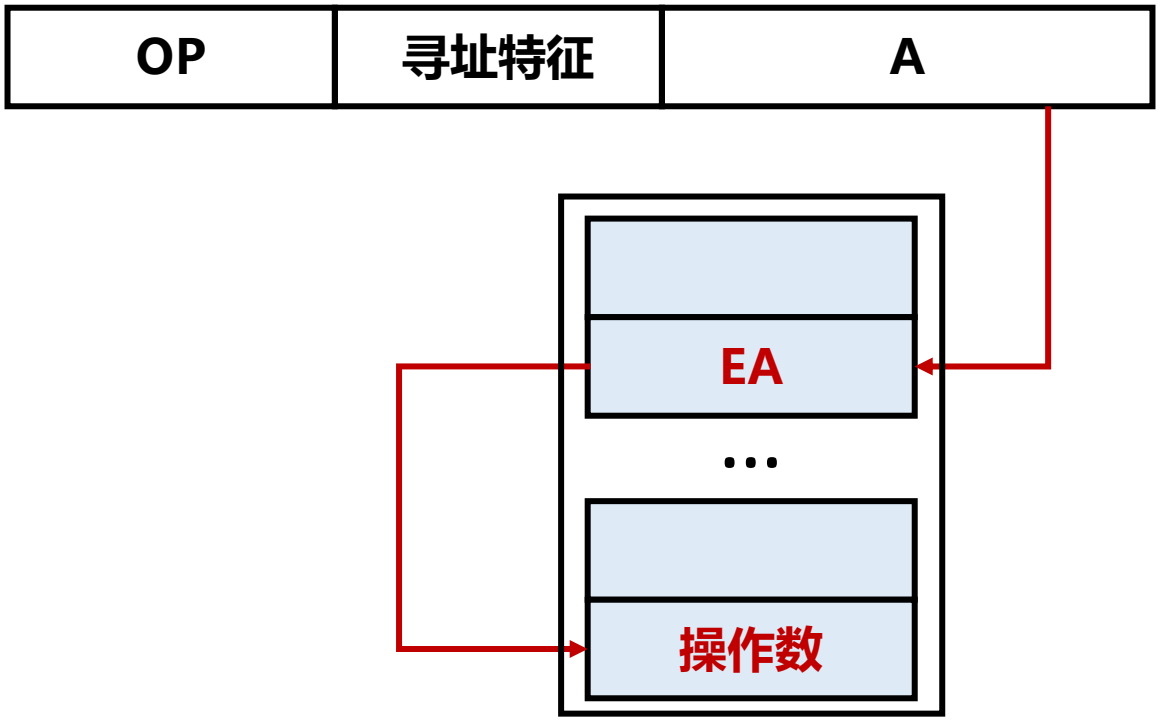
直接寻址

- **方式**：地址字段含有操作数的有效地址
- **用法**：早期计算机常用，在当代计算机体系结构中不多见
- **算法**：EA = A
- **优点**：只要求1次存储器访问，且无需为生成地址而专门计算
- **缺点**：有限的地址空间



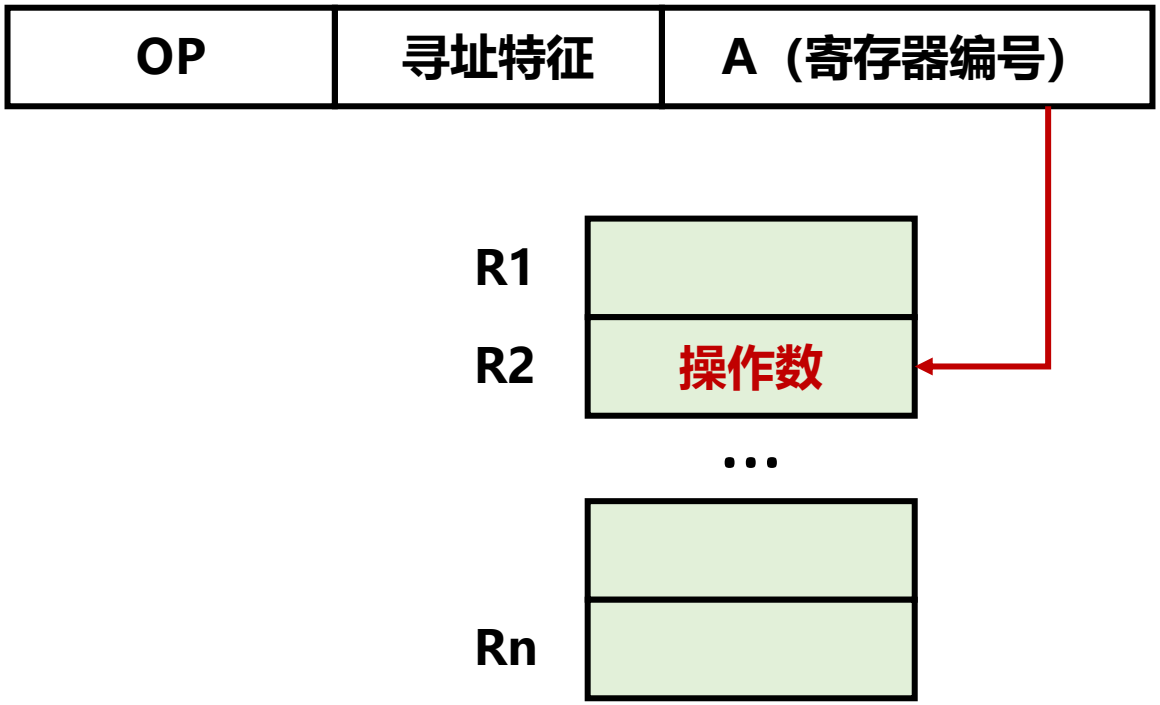
间接寻址

- **方式**：地址字段指示一个存储器字地址，而此地址出保存有操作数的全长度地址
- **算法**：EA = (A)
- **优点**：扩大了地址空间
- **缺点**：取操作数需要2次访问存储器
- **解释**：地址引用的数量限制可能是有益的



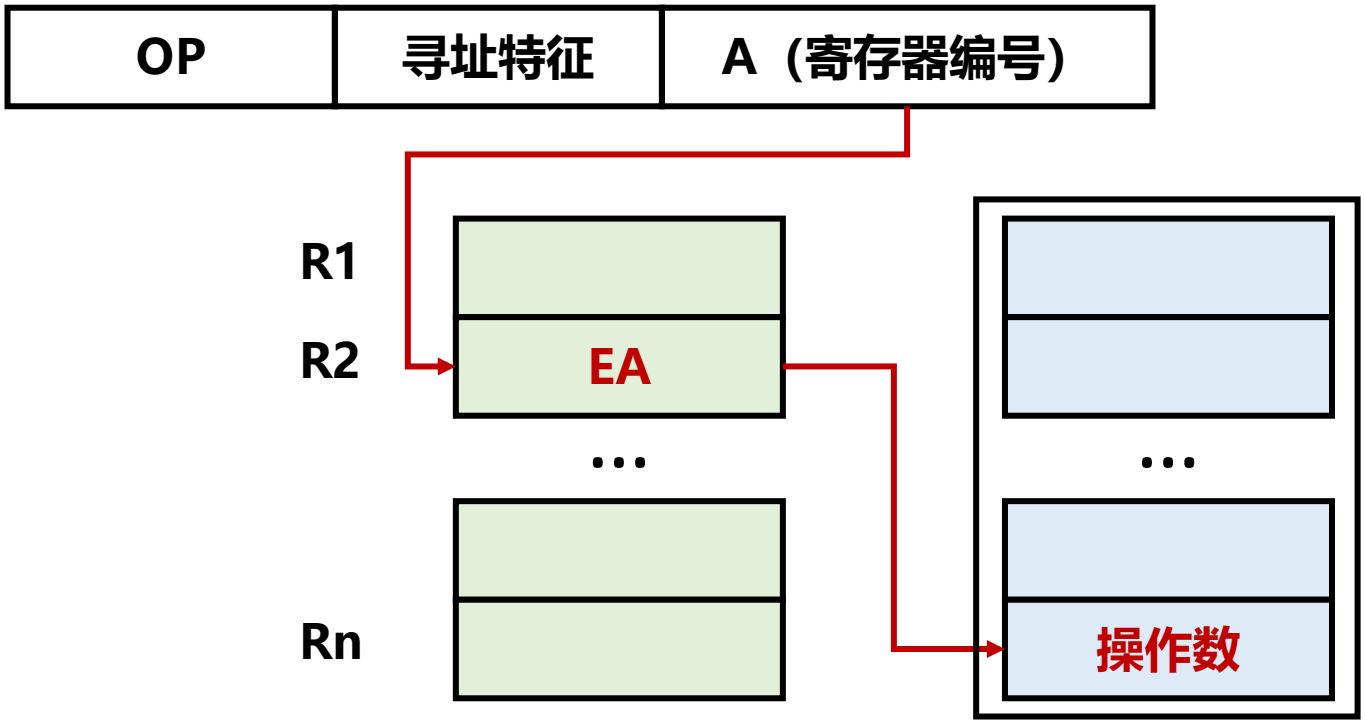
寄存器寻址

- **方式**：地址字段指示的是寄存器
- **算法**：EA = R
- **优点**：指令中仅需要一个较小的地址字段，且不需要存储器访问
- **缺点**：地址空间十分有限
- **解释**：寄存器寻址只有在被有效使用的时候才有意义



寄存器间接寻址

- **方式**：地址字段指示寄存器
- **算法**：EA = (R)
- **优点**：扩大了地址空间，比间接寻址少1次存储器访问
- **缺点**：相对于寄存器寻址，需要多1次存储器访问

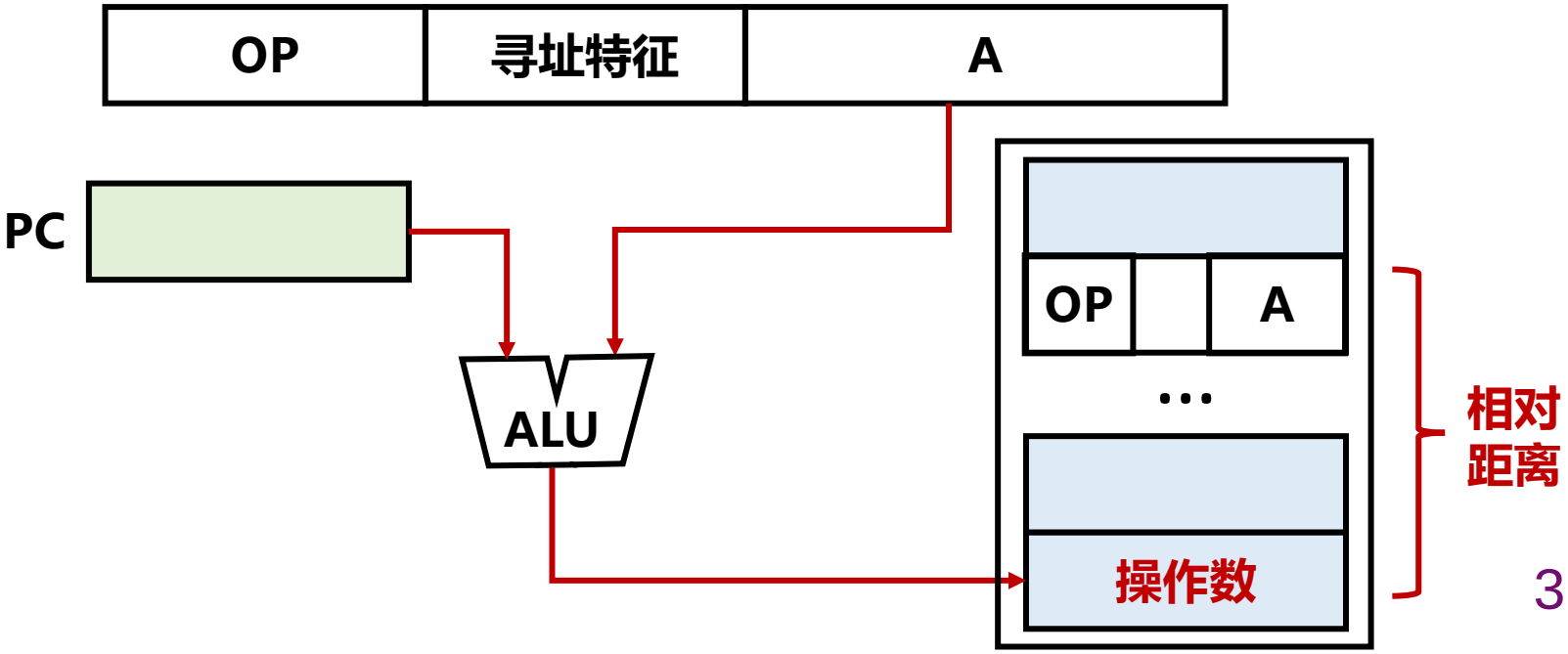


偏移寻址

- **方式**：结合直接寻址和寄存器间接寻址能力
- **算法**：EA = (R) + A
- **类型**
  - 相对寻址
  - 基址寄存器寻址
  - 变址寻址
- **解释**：偏移寻址要求指令有两个地址字段，至少其中一个是显式的

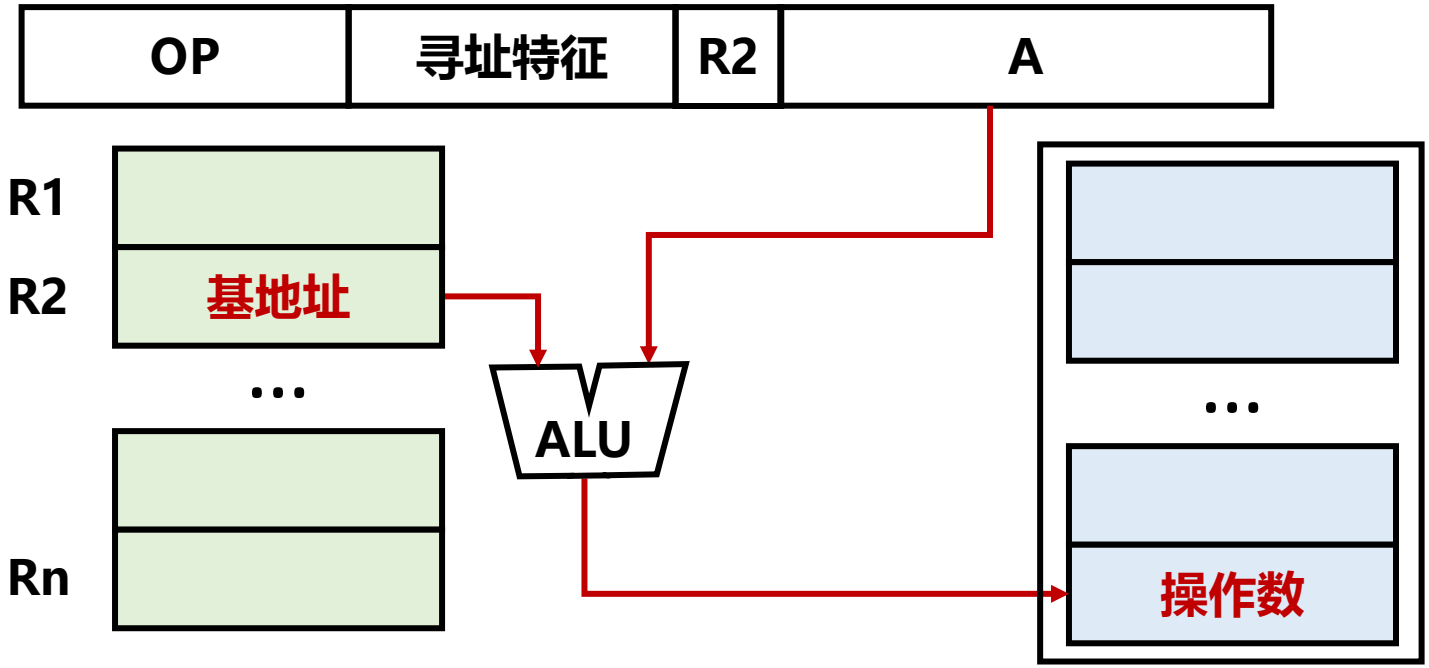
偏移寻址：相对寻址

- **方式**：隐含引用的寄存器是程序计数器（PC）
  - 此指令后续的下一条指令的地址加上地址字段的值产生有效地址
- **用法**：大多数存储器访问都相对靠近正在执行的指令，相对寻址可节省指令中的地址位数；可用于转移控制指令。
- **算法**：EA = (PC) + A
- **优点**：利用程序局部性原理，节省指令中地址的位数



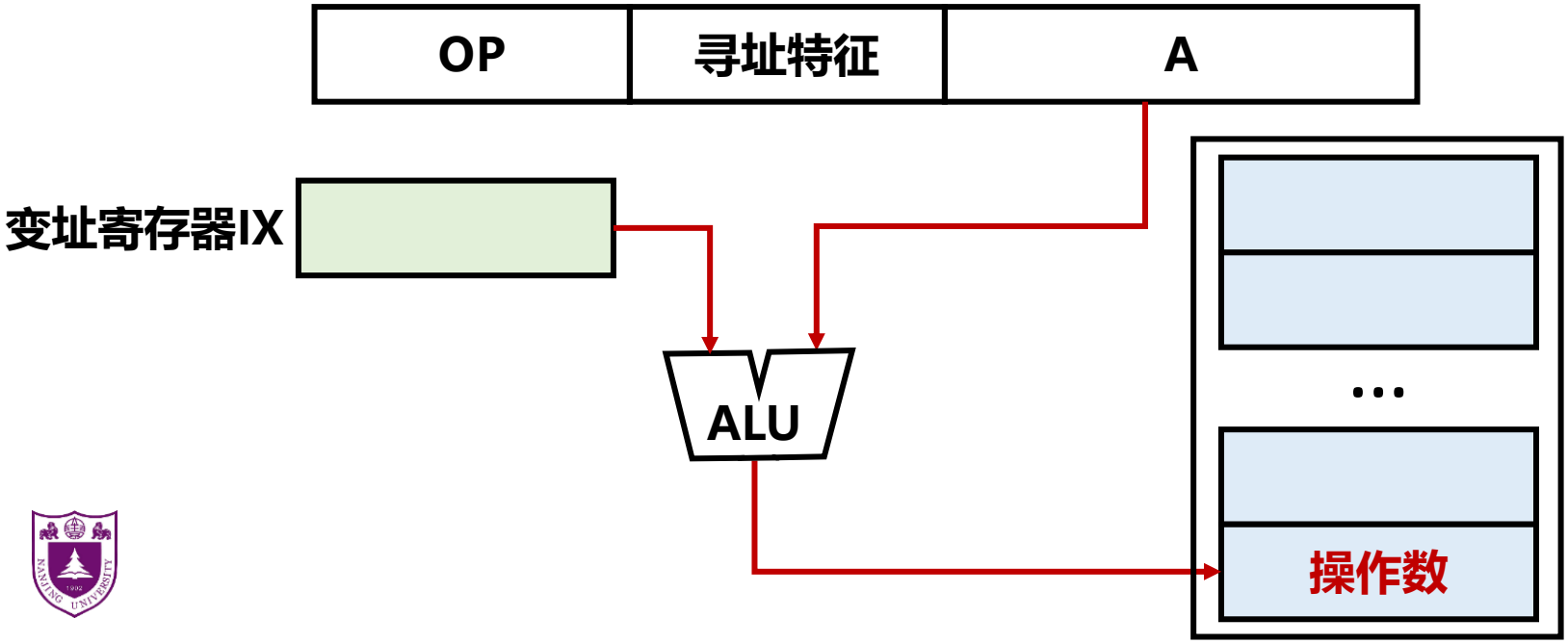
偏移寻址：基址寄存器寻址

- **方式**：被引用的寄存器含有一个存储器地址，地址字段含有一个相对于那个地址的偏移量（通常是无符号整数表示）
  - 寄存器引用可以是显式的，也可以是隐式的
- **算法**：EA = (B) + A
- **用法**：虚拟内存空间中的程序重定位



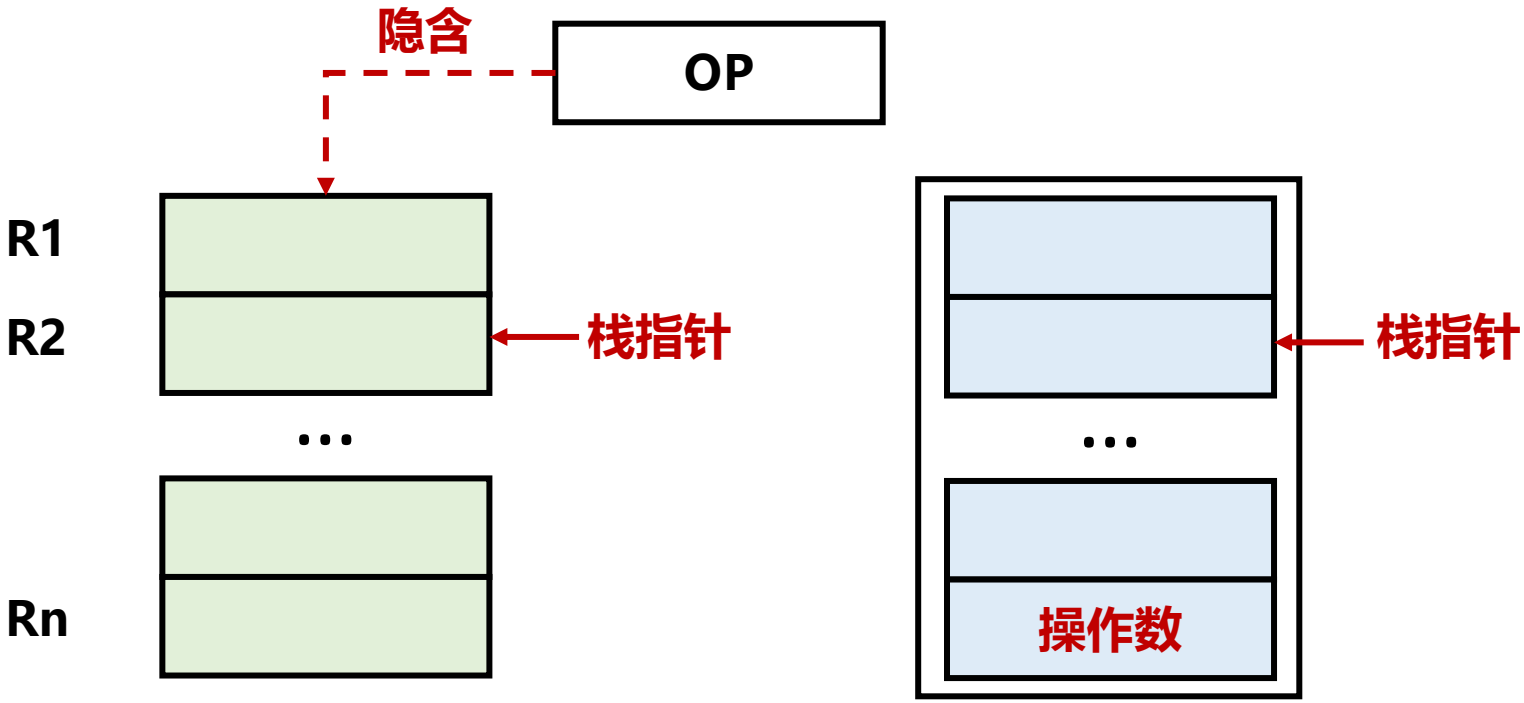
偏移寻址：变址寻址

- **方式**：指令地址字段引用一个主存地址，被引用的寄存器含有对于该地址的一个正的偏移量
- **算法**：EA = A + (IX)
- **用法**：为完成重复操作提供一种高效机制
- **扩展**：结合间接寻址和变址寻址
  - 前变址：EA = (A + (IX))；后变址：EA = (A) + (IX)



栈寻址

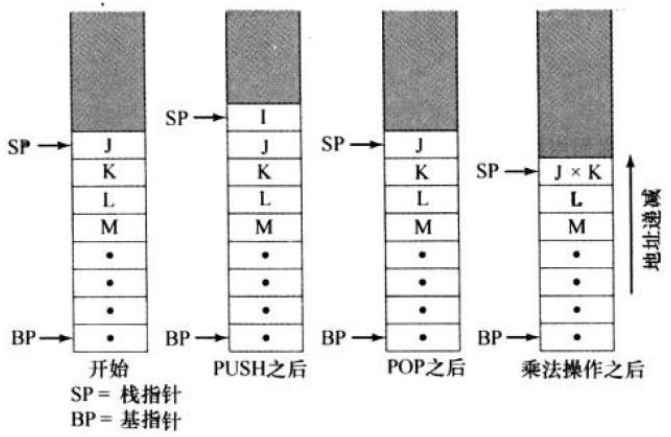
- **方式**：栈指针保存在寄存器中，对寄存器中栈位置的访问实际上是一种寄存器间接寻址方式
- **解释**：与栈相关的是一个指针，它的值是栈顶地址，或者当栈顶的两个元素已在CPU寄存器内，此时栈顶指针指向栈顶的第三个元素



栈

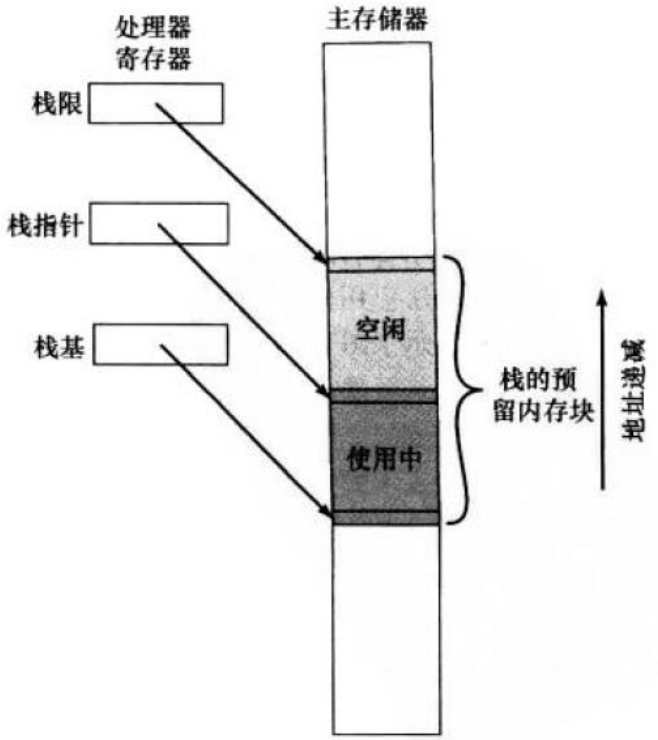
- **栈（stack）是有序元素组**，一次仅能存取它的一个元素
  - 栈顶（top）：存取元素的点
    - 栈的元素只能由栈顶添加或删除（后进先出）
  - 栈底（base）：栈中最后一个元素
- **栈的操作**

PUSH	栈顶上添加一个新元素
POP	移走栈顶元素
一元操作	对栈顶元素完成操作，以结果替换栈顶元素
二元操作	对栈顶元素操作，删除栈顶两元素，操作结果放到栈顶



栈（续）

- **栈的实现**
  - 栈基（base）：保存为栈保留的内存块底部位置的地址
  - 栈限（limit）：保存为栈保留的内存块另一端的地址
  - 栈指针（pointer）：保存栈顶的地址
  - 向上/向下增长：向着地址增大/减小的方向增长



栈（续）

- 栈的应用：表达式求值
  - 中缀（infix）表示 ➔ 后缀（postfix）/ 逆波兰（reverse polish）表示
    - 无论表达式多么复杂，后缀表示都不需要括号
  - 采用后缀表示，很容易使用栈来完成求值运算
    - 如果元素是变量或常数，则压入栈
    - 如果元素是操作符，则弹出栈顶的两个元素，完成运算后将结果压入栈

中缀	后缀
a + b	变成 ab +
a + (b × c)	变成 abc × +
(a + b) × c	变成 ab + c ×

输入	输出	栈（顶在右）
A+B×C+(D+E)×F	空	空
+B×C+(D+E)×F	A	空
B×C+(D+E)×F	A	+
×C+(D+E)×F	AB	+
C+(D+E)×F	AB	+ ×
+(D+E)×F	ABC	+ ×
(D+E)×F	ABC×	+
D+E)×F	ABC×	+(
+E)×F	ABC×D	+(
E)×F	ABC×D	+(+
)×F	ABC×DE	+(+
×F	ABC×DE+	+
F	ABC×DE+	+ ×
空	ABC×DE+F	+ ×
空	ABC×DE+F×	+ 空

图 10-17　由中缀表示法到后缀表示法的算式转换

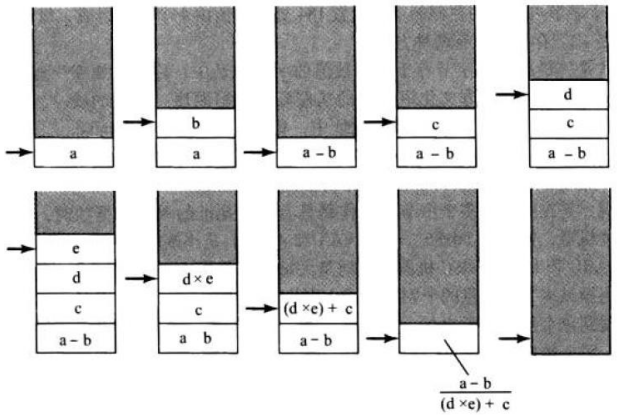


图 10-16　计算 f = (a - b)/[(d × e) + c]时栈的使用

回顾：指令格式

- 指令格式通过它的各个构成部分来定义指令的位安排
- 一个指令格式必须包含一个操作码，以及隐式或显式的、零个或多个操作数
- 指令格式必须显式或隐式地为每个操作数指定其寻址方式
- **大多数指令集使用不止一种指令格式**

指令格式的设计原则

- 指令尽量短
  - 程序占用存储空间小
- 有足够的操作码位数
  - 要为操作类型不断增加预留
- 操作码的编码必须有唯一的解释
  - 操作码译码时要么是唯一的合法编码，要么是不合法的
- 指令长度是字节的整数倍
  - 与内存按照字节寻址相对应，便于指令的读取和地址计算
- 合理选择地址字段的个数
  - 涉及到指令长度和规整性，是空间和时间开销权衡的结果
- 指令尽量规整
  - 简化硬件的实现

指令长度

- 最明显的权衡考虑是在强有力的指令清单和节省空间之间进行
  - 编程人员希望更多的操作码、更多的操作数、更多的寻址方式和更大的地址范围
  - 指令长度变短可以节省存储空间和减少数据传送时间
- 指令长度应该是字符长度或定点数长度的整数倍（比如n个字节）
- 指令长度应该等于存储器的传送长度（即数据总线宽度），或者这两个值其中之一是另一个的整数倍

位的分配

- 对于给定的指令长度，在操作码数目和寻址能力之间存在着权衡考虑
- 使用变长的操作码
  - 使用一个最小操作码长度，但是对于某些操作码，可通过使用指令附加位的方法来指定附加的操作
- 使用寻址位的考虑因素
  - 支持的寻址方式的种数
  - 操作数的数量：数量多，则操作数的位数短
  - 寄存器与存储器比较：能用于操作数引用的寄存器越多，指令需要的位数越少
  - 寄存器组的数目：对于固定数目的寄存器，功能上的分开将使指令只需较少的位数
  - 地址范围：比如间接寻址可以增大范围
  - 寻址粒度：同样大的寻址空间，使用较大的字时，需要的地址位更少

变长指令

- **提供不同长度的各种指令格式**
- **优点**
  - 易于提供大的操作码清单，而操作码具有不同的长度
  - 寻址方式能更灵活，指令格式能将各种寄存器和存储器引用加上寻址方式予以组合
- **缺点**
  - 增加了CPU的复杂程度（效率也会降低）
- **取至少等于最长指令长度的几个字节或几个字**

指令集设计

- **指令集的设计是件很复杂的事情，影响计算机系统的诸多方面**
  - 指令集定义了处理器应完成的多数功能，对处理器的实现有着显著的影响
  - 指令集是程序员控制处理器的方式，设计时必须考虑程序员的要求
- **设计的基本原则**
  - 完备性/完整性：操作类型应当尽可能完备，但太复杂了也会给硬件实现增加困难
  - 兼容性：应当兼容以前的指令系统，为软件重复利用带来方便
  - 均匀性：应当能对多种类型的数据进行处理
  - 可扩充性：操作码要预留一定的编码空间

指令集设计（续）

- **设计的基本问题**
  - 操作指令表：应提供多少和什么样的操作，操作有多复杂
  - 数据类型：对哪几种数据类型完成操作
  - 指令格式：指令的位长度、地址数目、各个字段的大小等
  - 寄存器：能被指令访问的寄存器数目以及它们的用途
  - 寻址：寻址方式的种类以及有效地址的计算
  - 下一条指令地址的确定：通常通过PC寄存器实现

总结

- **指令**
  - 操作码
  - 操作数
    - 寻址方式：立即寻址，直接寻址，间接寻址，寄存器寻址，寄存器间接寻址，偏移寻址，栈寻址
- **指令格式**：指令长度，位分配，变长指令
- **指令集设计**

谢谢

bohanliu@nju.edu.cn