中国科学院大学
University of Chinese Academy of Sciences

B0911006Y-01 2024-2025学年春季学期

计算机组成原理

Principles of Computer Organization

指令系统 I

机器指令如何构成？地址码、操作数有哪些类型？

主讲教师：石 侃

shikan@ict.ac.cn

2025年4月23日

第7章 指令系统

The easy ride of software is over

- 很久很久以前，软件工程师可以不关心硬件，只关注程序功能的实现
- 就算现在的硬件跑不动，过一阵会出现更新、更强的硬件
  - *Andy & Bill's Law: Andy gives, Bill takes away*
  - *Jensen & Sam's Law: Jensen gives, Sam takes away*

The easy ride of software is over

- 很久很久以前，软件工程师可以不关心硬件，只关注程序功能的实现
- 就算现在的硬件跑不动，过一阵会出现更新、更强的硬件
  - *Andy & Bill's Law: Andy gives, Bill takes away*
  - *Jensen & Sam's Law: Jensen gives, Sam takes away*
- 随着摩尔定律走向终结，意味着软件工程师需要更多了解硬件
- 因此，软件必须认真考虑如何在硬件、特别是专用硬件上高效执行的问题
  - *Talk from Hennessy & Patterson, Mar 2017*

机器指令 vs 机器的指令系统

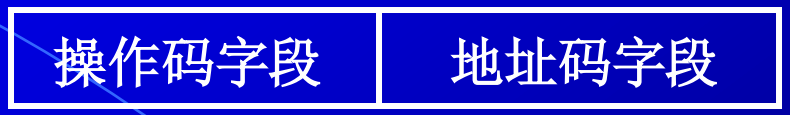
- 机器指令：每一条机器语言的语句
- 机器的指令系统
  - 所有机器指令的集合，反映了机器的性能
  - 指令系统处在软/硬件交界面，同时被硬件设计者和系统软件工程师看到
  - 指令系统的好坏，（很大程度上）决定了计算机的性能和成本
- 计算机设计师需要确定并用硬件实现机器的指令系统
- 计算机使用者（系统程序员）需要根据指令系统编写汇编程序
- 只有熟悉计算机硬件，才能写出高效的代码

回顾：冯诺依曼结构机器对指令的规定

- 指令用二进制表示，和数据一起存放在主存中
- 指令由两部分组成：操作码+操作数（或其地址码）
  - Operation Code (Opcode): defines the operation type
  - Operand: indicates the operation source and/or destination

7.1 机 器 指 令

一、指令的一般格式



1. 操作码　　反映机器做什么操作
位数决定指令条数

(1) 长度固定

用于指令字长较长的情况 ， RISC

如 IBM 370(指令字长16/32/48位） 操作码 8 位

(2) 长度可变

操作码分散在指令字的不同字段中

用于指令字长较短的情况，如Intel 8086（1-6字节）/ X86（1-15字节）

(3) 扩展操作码技术

7.1

操作码的位数随地址数的减少而增加

OP	A_1	A_2	A_3

4 位操作码

0000	A_1	A_2	A_3
0001	A_1	A_2	A_3
⋮	⋮	⋮	⋮
1110	A_1	A_2	A_3

最多15条三地址指令

8 位操作码

1111	0000	A_2	A_3
1111	0001	A_2	A_3
⋮	⋮	⋮	⋮
1111	1110	A_2	A_3

最多15条二地址指令

12 位操作码

1111	1111	0000	A_3
1111	1111	0001	A_3
⋮	⋮	⋮	⋮
1111	1111	1110	A_3

最多15条一地址指令

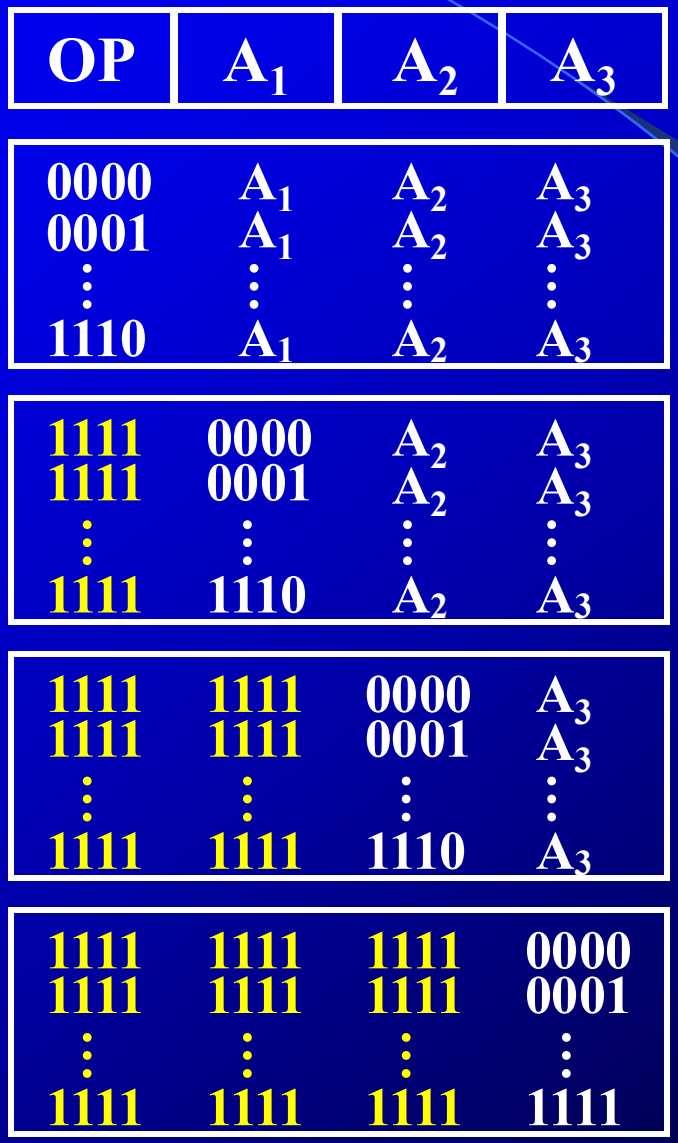
16 位操作码

1111	1111	1111	0000
1111	1111	1111	0001
⋮	⋮	⋮	⋮
1111	1111	1111	1111

16条零地址指令

(3) 扩展操作码技术

操作码的位数随地址数的减少而增加

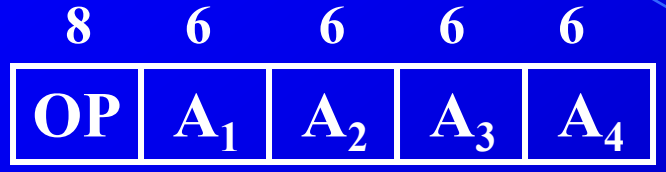


三地址指令操作码每减少一种可多构成 2^4 种二地址指令

二地址指令操作码每减少一种可多构成 2^4 种一地址指令

2. 地址码

(1) 四地址



A_1 第一操作数地址

A_2 第二操作数地址

A_3 结果的地址

A_4 下一条指令地址

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

设指令字长为 32 位

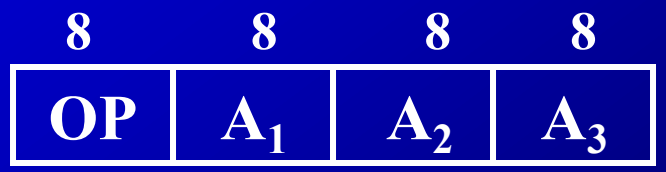
操作码固定为 8 位

4 次访存

寻址范围 $2^6 = 64$

若 PC 代替 A_4

(2) 三地址



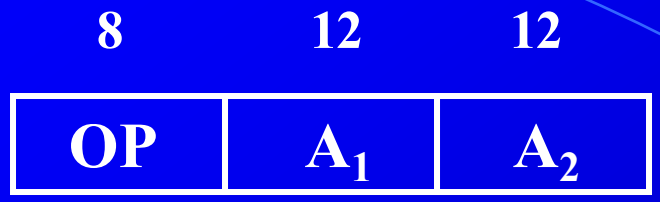
(A_1) OP $(A_2) \longrightarrow A_3$

4 次访存

寻址范围 $2^8 = 256$

若 A_3 用 A_1 或 A_2 代替

(3) 二地址



或 (A_1) OP $(A_2) \longrightarrow A_1$　　4 次访存

(A_1) OP $(A_2) \longrightarrow A_2$　　寻址范围 $2^{12} = 4$ K

若结果存于 ACC　3次访存　　若ACC 代替 A_1（或A_2）

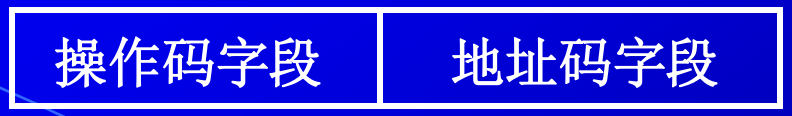
(4) 一地址

2 次访存

(ACC) OP $(A_1) \longrightarrow$ ACC　　寻址范围 $2^{24} = 16$ M

(5) 零地址　　无地址码

二、指令字长



指令字长决定于
- 操作码的长度
- 操作数地址的长度
- 操作数地址的个数

1. 指令字长 固定

指令字长 = 存储字长

2. 指令字长 可变

按字节的倍数变化

小结

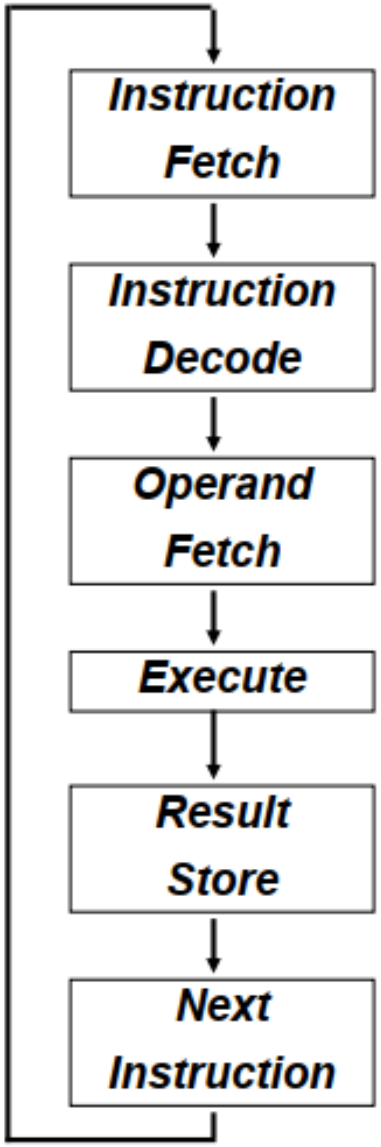
- 当用一些硬件资源代替指令字中的地址码字段后
  - 可扩大指令操作数的寻址范围
  - 可缩短指令字长
  - 可减少访存次数
- 当指令的地址字段为寄存器时

  三地址　OP　R_1，R_2，R_3

  二地址　OP　R_1，R_2

  一地址　OP　R_1
  - 可缩短指令字长
  - 指令执行阶段不访存

从指令执行周期看指令设计涉及的问题



从存储器取指令

指令地址、指令长度（定长/变长）

对指令译码，以确定要做什么操作

指令格式、操作码编码、操作数类型

计算操作数地址并取操作数

地址码、寻址方式、操作数格式和存放方式

执行计算、并得到标志位

操作类型、标志或条件码

将计算结果保存到目的地

结果数据位置（目的操作数）

计算下条指令地址（通常和取指同时进行）

下条指令地址（顺序/转移）

7.2 操作数类型和操作种类

一、操作数类型

地址	无符号整数
数字	定点数、浮点数、十进制数
字符	ASCII
逻辑数	逻辑运算

二、数据在存储器中的存放方式

32位数：

3	2	1	0

字地址

3	2	1	0	0
7	6	5	4	4

低字节 在 低地址
小尾端(Little Endian)

字地址

0	1	2	3	0
4	5	6	7	4

高字节 在 低地址
大尾端（Big Endian)

Endianness (字节序)

- “endian”一词来源于乔纳森·斯威夫特(Jonathan Swift)的1726年讽刺小说《格列佛游记》Gulliver's Travels
- 小说中，小人国为“水煮蛋”该从大的一端（Big-End）剥开还是小的一端（Little-End）剥开而争论，争论的双方分别被称为“大端派”和“小端派”
  - 格列佛发现Lilliput国家当今统治者的祖父当年按古法打鸡蛋时碰巧将一个手指弄破了，因此曾立下一法，要其所有公民吃蛋时都要从小头开剥。那些以往从大头开剥的公民理所当然地感到愤怒，内战爆发了。结果，大头开剥者被赶到了附近的一座岛上，成立了Blefuscu王国。
  - Lilliput曾经发生过6次叛乱，其中一个皇帝送了命，另一个丢了王位。这些叛乱大多都是由Blefuscu的国王大臣们煽动起来的。叛乱平息后，流亡的人总是逃到那个Blefuscu帝国去寻求避难。据估计，先后几次有11000人情愿受死也不肯去打破鸡蛋较小的一端。
  - 在那个时代，Swift是在讽刺英国和法国之间的持续冲突

- 1980年，Danny Cohen(一位网络协议的早期开发者)，在其论文"On Holy Wars and a Plea for Peace"中，为平息一场关于字节该以什么样的顺序传送的争论，而第一次引用了该词

BIG Endian versus Little Endian

Ex1: Memory layout of a number ABCDH located in 1000

In Big Endian: →

CD	1001
AB	1000

In Little Endian: →

AB	1001
CD	1000

Ex2: Memory layout of a number 00ABCDEFH located in 1000

In Big Endian: →

00
AB
CD
EF

In Little Endian: →

00
AB
CD
EF

BIG Endian versus Little Endian

Ex1: Memory layout of a number ABCDH located in 1000

In Big Endian: →

CD	1001
AB	1000

In Little Endian: →

AB	1001
CD	1000

Ex2: Memory layout of a number 00ABCDEFH located in 1000

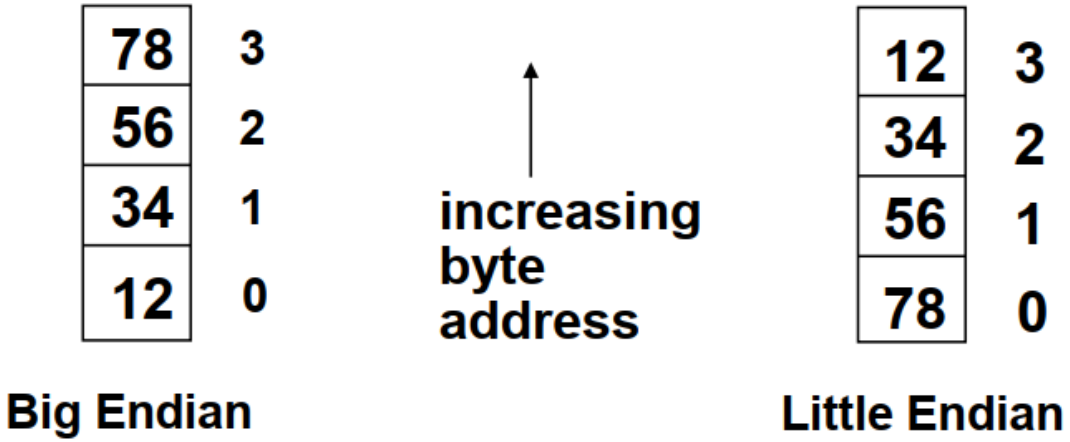
In Big Endian: →

00	1000
AB	1001
CD	1002
EF	1003

In Little Endian: →

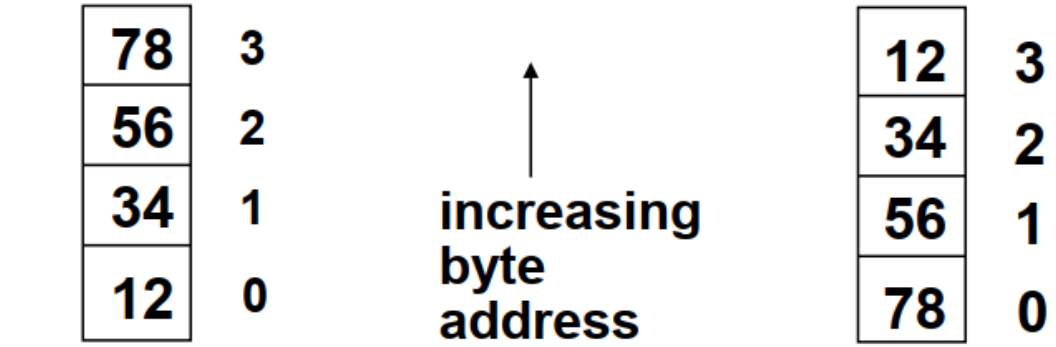
00	1003
AB	1002
CD	1001
EF	1000

Byte Swap Problem（字节交换问题）



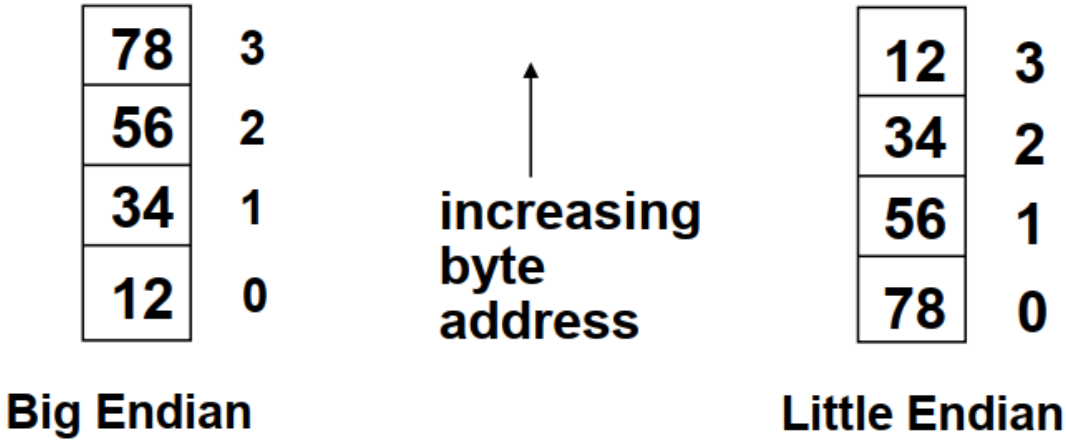
上述存放在0号单元的数据（字）是什么？

Byte Swap Problem（字节交换问题）



上述存放在0号单元的数据（字）是什么？12345678H？ 78563412H？

Byte Swap Problem（字节交换问题）

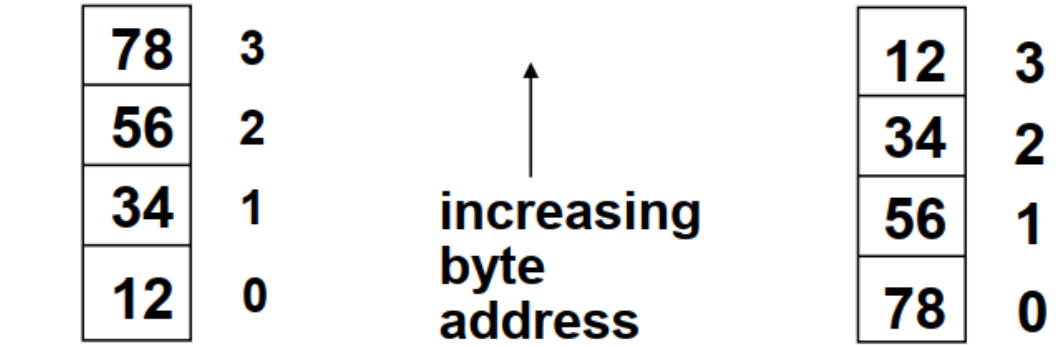


上述存放在0号单元的数据（字）是什么？12345678H？ 78563412H？

存放方式不同的机器间程序移植或数据通信时，会发生什么问题？

- 每个系统内部是一致的，但在系统间通信时可能会发生问题！
- 因为顺序不同，需要进行顺序转换

Byte Swap Problem（字节交换问题）

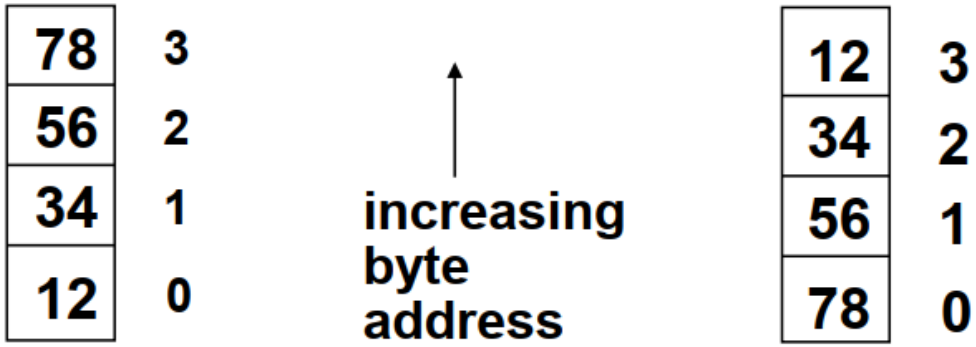


上述存放在0号单元的数据（字）是什么？12345678H？ 78563412H？

存放方式不同的机器间程序移植或数据通信时，会发生什么问题？

- 每个系统内部是一致的，但在系统间通信时可能会发生问题！
- 因为顺序不同，需要进行顺序转换
- C/C++程序中数据存储顺序与编译平台所在的CPU相关，而Java程序则唯一采用big endian方式来存储数据

Byte Swap Problem（字节交换问题）



上述存放在0号单元的数据（字）是什么？12345678H？ 78563412H？

存放方式不同的机器间程序移植或数据通信时，会发生什么问题？

- 每个系统内部是一致的，但在系统间通信时可能会发生问题！
- 因为顺序不同，需要进行顺序转换
- C/C++程序中数据存储顺序与编译平台所在的CPU相关，而Java程序则唯一采用big endian方式来存储数据

音、视频和图像等文件格式或处理程序都涉及到字节顺序问题

ex. Little endian: GIF, PC Paintbrush, Microsoft RTF,etc

Big endian: Adobe Photoshop, JPEG, MacPaint, etc

Alignment(对齐)

Alignment: 要求数据的地址是相应的边界地址

- 目前机器字长(寄存器位数)一般为32位或64位，而存储器地址按字节编址
- 指令系统支持对字节、半字、字及双字的运算，也有位处理指令

Alignment(对齐)

Alignment: 要求数据的地址是相应的边界地址

- 目前机器字长(寄存器位数)一般为32位或64位，而存储器地址按字节编址
- 指令系统支持对字节、半字、字及双字的运算，也有位处理指令
- 各种不同长度的数据存放时，有两种处理方式:
  - 按边界对齐 （假定存储字的宽度为32位，按字节编址）
    - 字地址：4的倍数(地址低两位为0)
    - 半字地址：2的倍数(地址低位为0)
    - 字节地址：任意

每4个字节可同时读写

Alignment(对齐)

Alignment: 要求数据的地址是相应的边界地址

- 目前机器字长(寄存器位数)一般为32位或64位，而存储器地址按字节编址
- 指令系统支持对字节、半字、字及双字的运算，也有位处理指令
- 各种不同长度的数据存放时，有两种处理方式:
  - 按边界对齐 （假定存储字的宽度为32位，按字节编址）
    - 字地址：4的倍数(地址低两位为0)
    - 半字地址：2的倍数(地址低位为0)
    - 字节地址：任意

    每4个字节可同时读写
  - 不按边界对齐

    坏处：可能会增加访存次数！

    （在学习完《存储器》一章后会更清楚此问题！）

Alignment(对齐)

存储器按字节编址

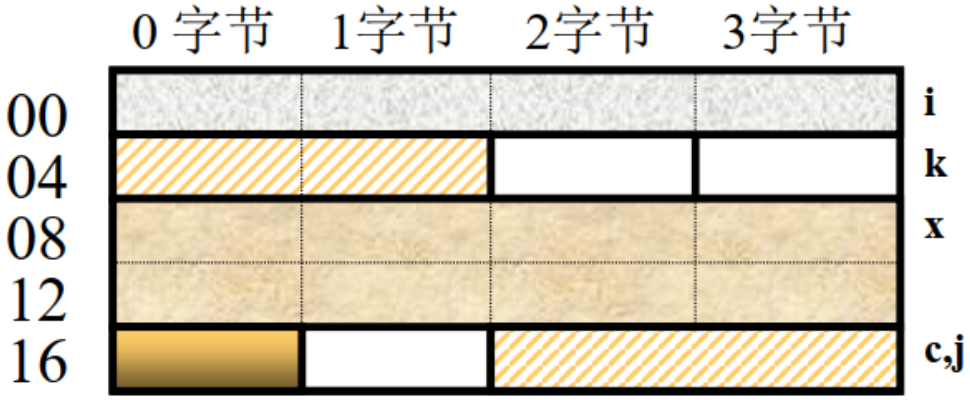
每次只能读写某个字地址开始的4个单元中连续的1个、2个、3个或4个字节（对齐规则：K字节大小的对象，起始地址为K的整数倍。例如，Linux下short型数据的地址需是2的倍数，int等类型是4的倍数，char类型数据是1字节的整数倍对齐）

如：int i, short k, double x, char c, short j,……

按边界对齐

x：2个周期

j：1个周期



则：&i=0; &k=4; &x=8; &c=16; &j=18;……

参考1：https://en.wikipedia.org/wiki/Data_structure_alignment

参考2：《深入理解计算机系统》第3.9.3小节：数据对齐

Alignment(对齐)

存储器按字节编址

每次只能读写某个字地址开始的4个单元中连续的1个、2个、3个或4个字节（对齐规则：K字节大小的对象，起始地址为K的整数倍。例如，Linux下short型数据的地址需是2的倍数，int等类型是4的倍数，char类型数据是1字节的整数倍对齐）

虽节省了空间，但增加了访存次数！

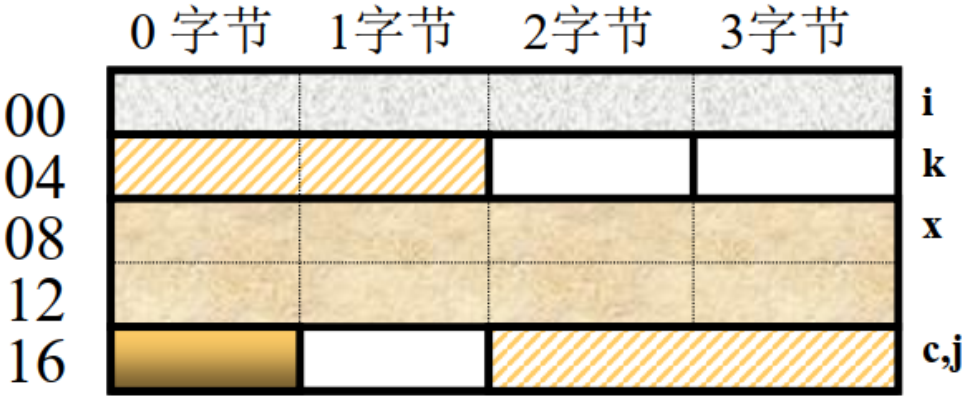
需要权衡，目前来看，浪费一点存储空间没有关系！

如：int i, short k, double x, char c, short j,……

按边界对齐

x：2个周期

j：1个周期

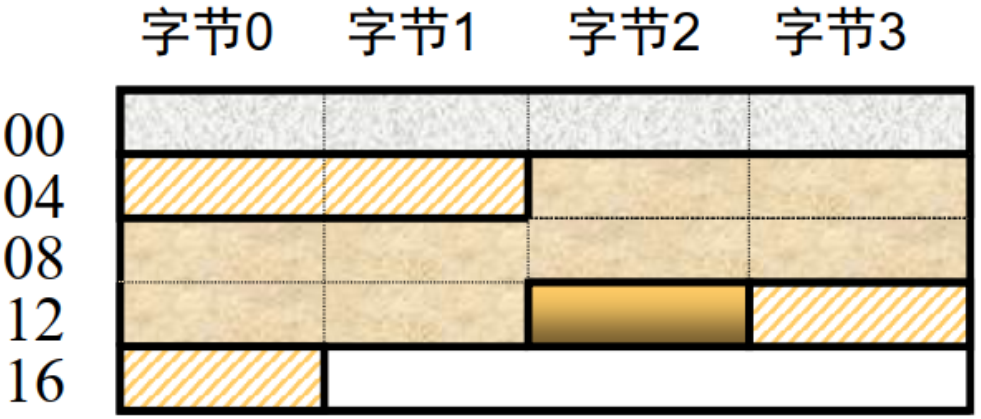


则：&i=0; &k=4; &x=8; &c=16; &j=18;……

边界不对齐

x：3个周期

j：2个周期



则： &i=0; &k=4; &x=6; &c=14; &j=15;……

参考1：https://en.wikipedia.org/wiki/Data_structure_alignment

参考2：《深入理解计算机系统》第3.9.3小节：数据对齐

存储器中的数据存放（存储字长为32位）

7.2

边界对准（aligned）

				地址（十进制）
字（地址 0）				0
字（地址 4）				4
字节（地址11）	字节（地址10）	字节（地址 9）	字节（地址 8）	8
字节（地址15）	字节（地址14）	字节（地址13）	字节（地址12）	12
半字（地址18）✓		半字（地址16）✓		16
半字（地址22）✓		半字（地址20）✓		20
双字（地址24）▲				24
双字				28
双字（地址32）▲				32
双字				36

边界未对准（unaligned）

			地址（十进制）
字(地址2)		半字(地址0)	0
字节(地址7)	字节(地址6)	字(地址4)	4
半字(地址10)		半字(地址8)	8

复习：

1. 指令与指令集
2. 指令的一般格式：操作码+地址码（操作数）
3. 指令字长
4. 扩展操作码：操作码的位数随地址数的减少而增加
5. 指令各个部分在指令执行周期内（指令的生命周期内）的作用和功能
6. 操作数类型
7. 数据在存储器中的存放
   - ① 程序中数据类型宽度
   - ② 字在存放时内部字节之间的顺序：大端BE与小端LE
   - ③ **The representation of dates on different areas on the planet is subjected to the same endianness, but instead of being about bytes, is about days, months and years:**
     - **China: big-endian representation: yyyy-MM-dd, e.g. 2023-04-10**
     - **US: middle-endian representation: MM-dd-yyyy, e.g. 04-10-2023**
     - **Europe: little-endian representation: dd-MM-yyyy, e.g. 10-04-2023**
     - **Japan(和历)：大端，e.g. 令和5年，R050410**
   - ④ 字在存放时的边界对齐
     - 规则：K字节大小的对象，起始地址为K的整数倍

中国科学院大学
University of Chinese Academy of Sciences

B0911006Y-01 2024-2025学年春季学期

计算机组成原理

Principles of Computer Organization

指令系统 II

指令中的操作类型、机器指令的寻址方式

主讲教师：石 侃
shikan@ict.ac.cn

2025年4月30日

三、操作类型：机器有哪些动作

7.2

1. 数据传送

源	寄存器	寄存器	存储器	存储器
目的	寄存器	存储器	寄存器	存储器
例如	MOVE	STORE MOVE PUSH	LOAD MOVE POP	MOVE

置“1”，清“0”

2. 算术逻辑操作

加、减、乘、除、增 1、减 1、求补（取负数）、浮点运算、十进制运算

与、或、非、异或、位操作、位测试、位清除、位求反

如 8086　ADD SUB MUL DIV INC DEC CMP NEG（求补）
AAA（加法后的非压缩BCD码调整） AAS AAM AAD
AND OR NOT XOR TEST

3. 移位操作

算术移位　逻辑移位

循环移位（带进位和不带进位）

4. 转移

(1) 无条件转移　JMP X

(2) 条件转移（Branch）

结果为零转　（Z = 1）JZ

结果溢出转　（O = 1）JO

结果有进位转（C = 1）JC

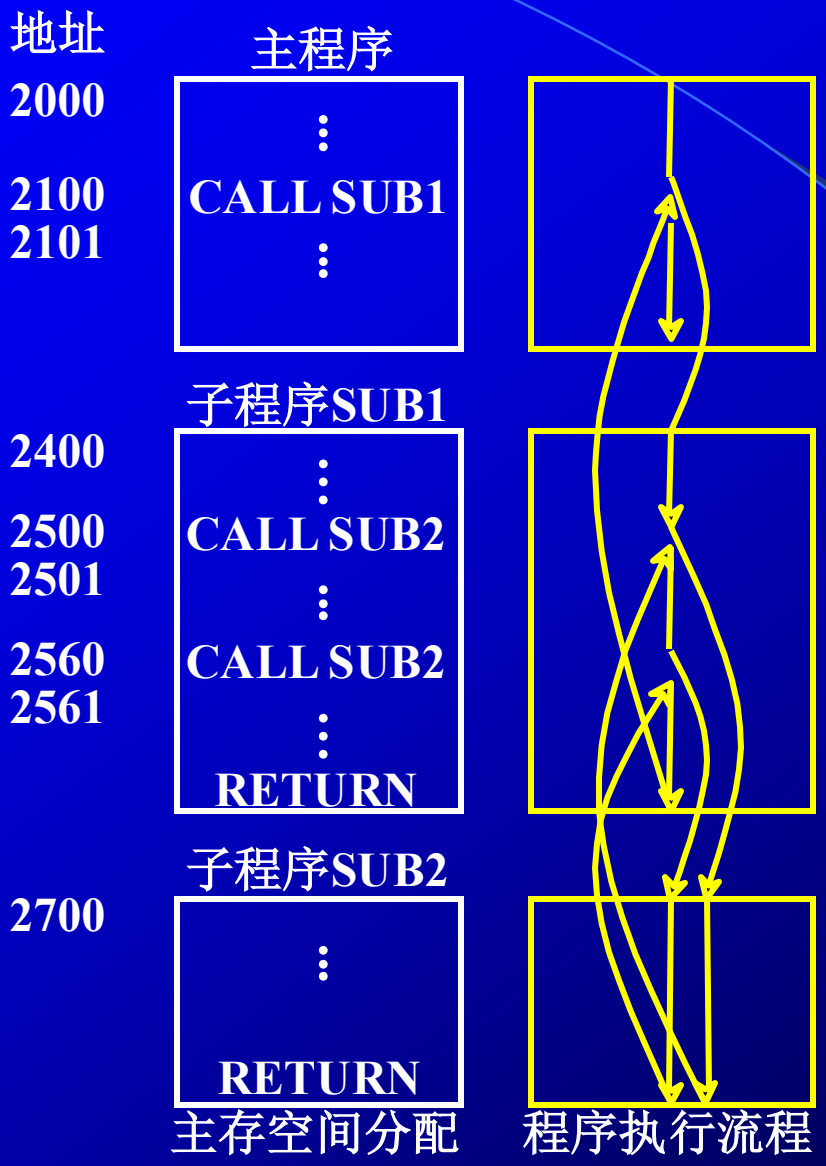
跳过一条指令 SKP

如

```
300
 ⋮
305  SKP  DZ  D = 0 则跳
306
307
```

完成触发器

(3) 调用和返回


地址
主程序
2000
2100
CALL SUB1
2101
子程序SUB1
2400
2500
CALL SUB2
2501
2560
CALL SUB2
2561
RETURN
子程序SUB2
2700
RETURN
主存空间分配
程序执行流程

(4) 陷阱（Trap）与陷阱指令

意外事故的中断

- 一般不提供给用户直接使用

  在出现事故时，由 CPU 自动产生并执行（隐指令）

- 设置供用户使用的陷阱指令

  如 8086　　INT　TYPE　　软中断

  提供给用户使用的陷阱指令，完成系统调用

5. 输入输出

入　　端口地址 ⟶ CPU 的寄存器

如　　IN AX, *m*　　IN AX, DX

出　　CPU 的寄存器 ⟶ 端口地址

如　　OUT *n*, AX　　OUT DX, AX

7.3 寻址方式

寻址方式　确定 本条指令 的 操作数地址

下一条 欲执行 指令 的 指令地址

寻址方式
- 指令寻址
- 数据寻址

7.3 寻址方式

一、指令寻址

顺序　　(PC) + 1 ⟶ PC

跳跃　　由转移指令指出

二、数据寻址

7.3

操作码	寻址特征	形式地址 A

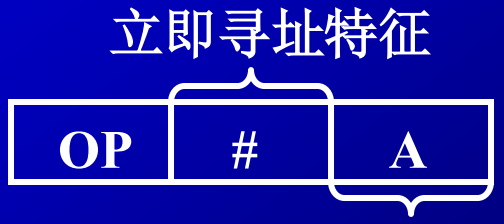
形式地址(A)　　指令字中的地址（地址码字段）

有效地址（EA,Effective Address)　操作数的真实地址

约定　指令字长 = 存储字长 = 机器字长

1. 立即寻址（Immediate Addressing）

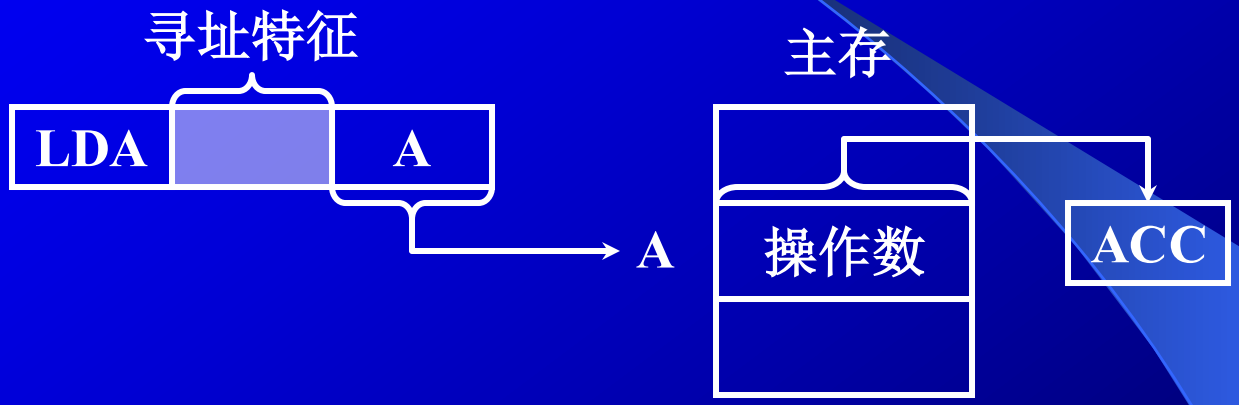
形式地址 A 就是操作数



- 指令执行阶段不访存
- A 的位数限制了立即数的范围

2. 直接寻址

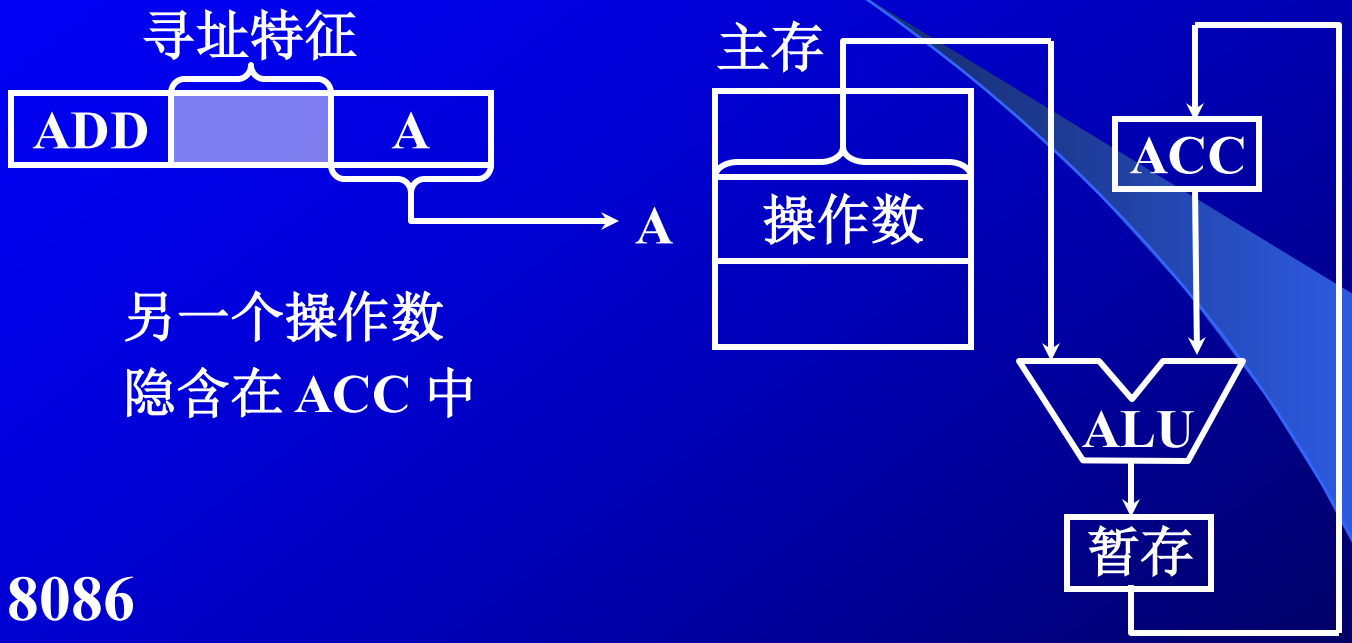
EA = A　　有效地址由形式地址直接给出



- 执行阶段访问一次存储器
- 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（必须修改A）

3. 隐含寻址

操作数地址隐含在操作码中



如 8086

MUL 指令　被乘数隐含在 AX（16位）或 AL（8位）中

MOVS 指令　源操作数的地址隐含在 SI 中

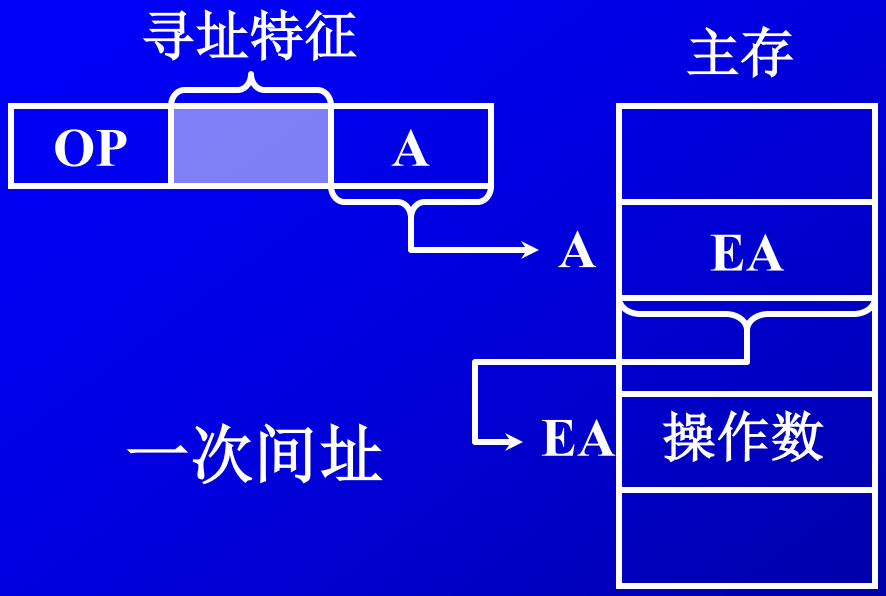
目的操作数的地址隐含在 DI 中

- 指令字中少了一个地址字段，可缩短指令字长

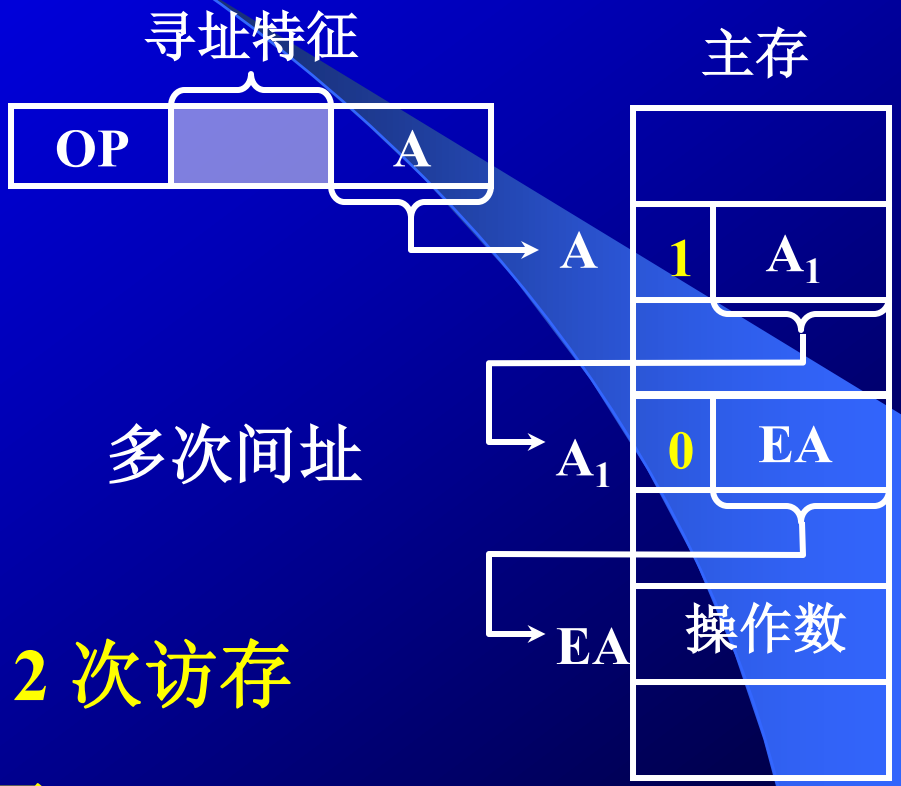
4. 间接寻址

7.3

EA＝（A）　有效地址由形式地址间接提供



一次间址

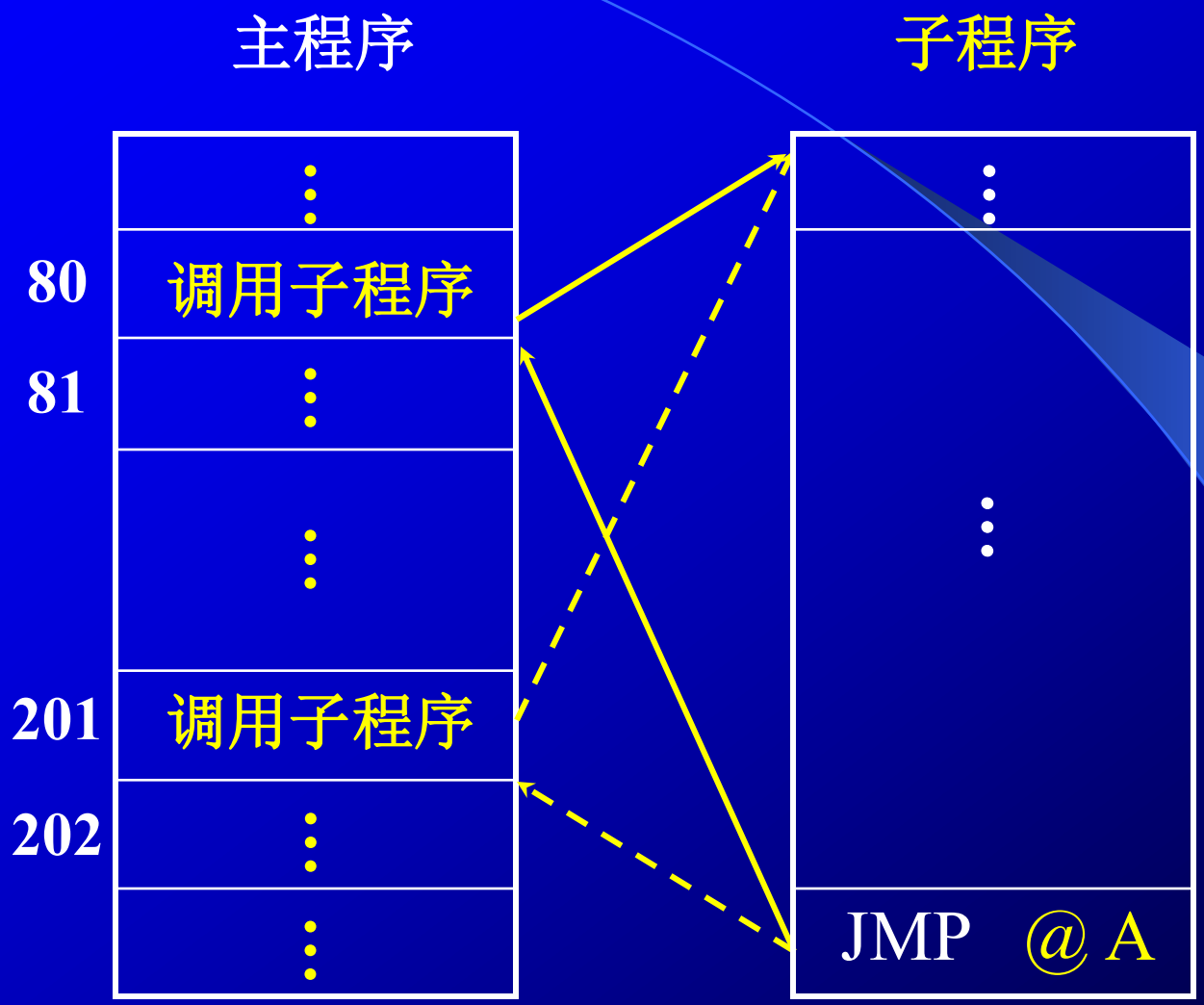


多次间址

- 执行指令阶段 2 次访存
- 可扩大寻址范围
- 便于编制程序

多次访存

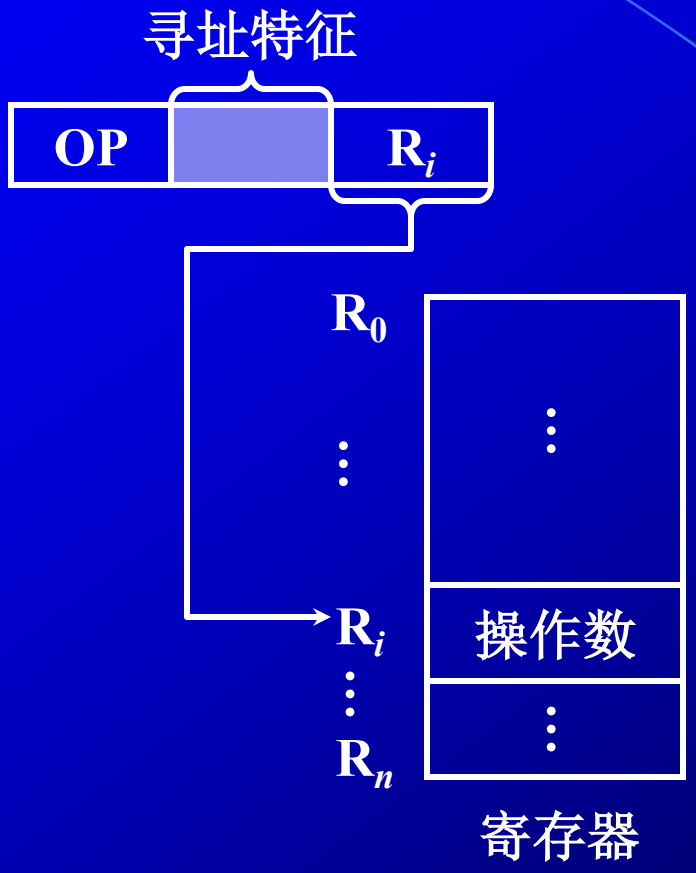
间接寻址编程举例



@ 间址特征

5. 寄存器寻址

$EA = R_i$　　有效地址即为寄存器编号

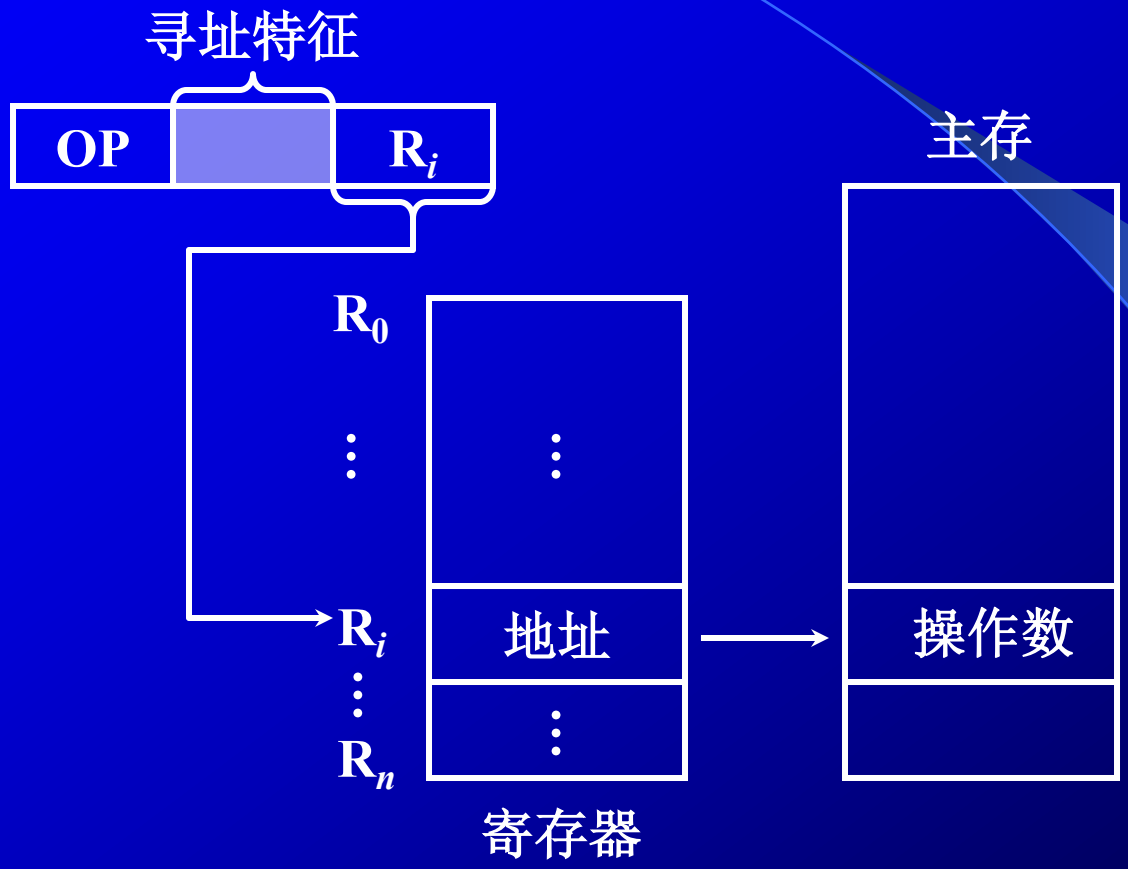


- 执行阶段不访存，只访问寄存器，执行速度快
- 寄存器个数有限，可缩短指令字长

6. 寄存器间接寻址

7.3

$EA = (R_i)$　　有效地址在寄存器中

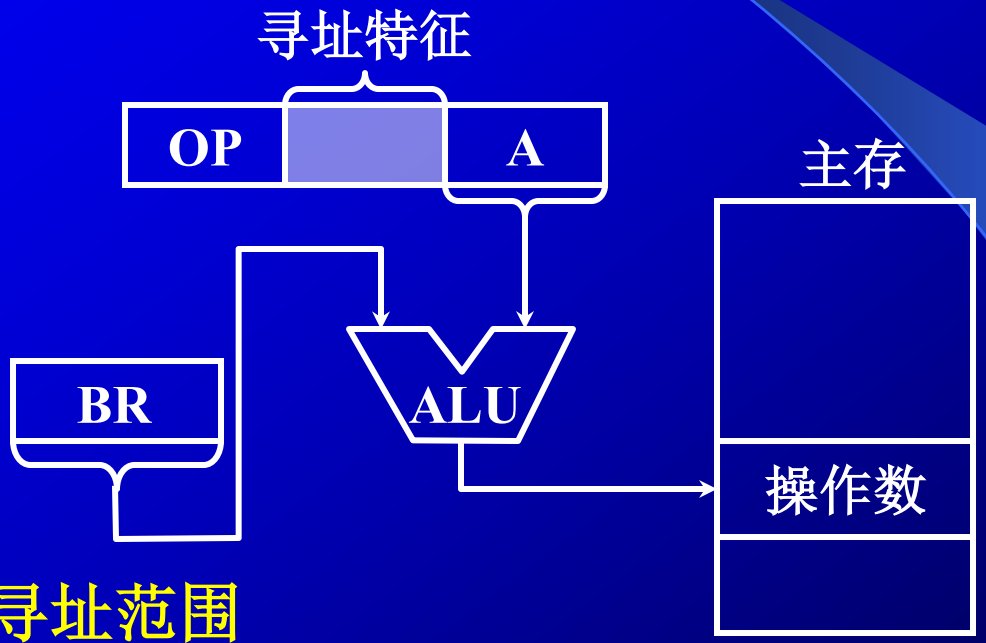


- 有效地址在寄存器中， 操作数在存储器中， 执行阶段访存
- 便于编制循环程序

7. 基址寻址

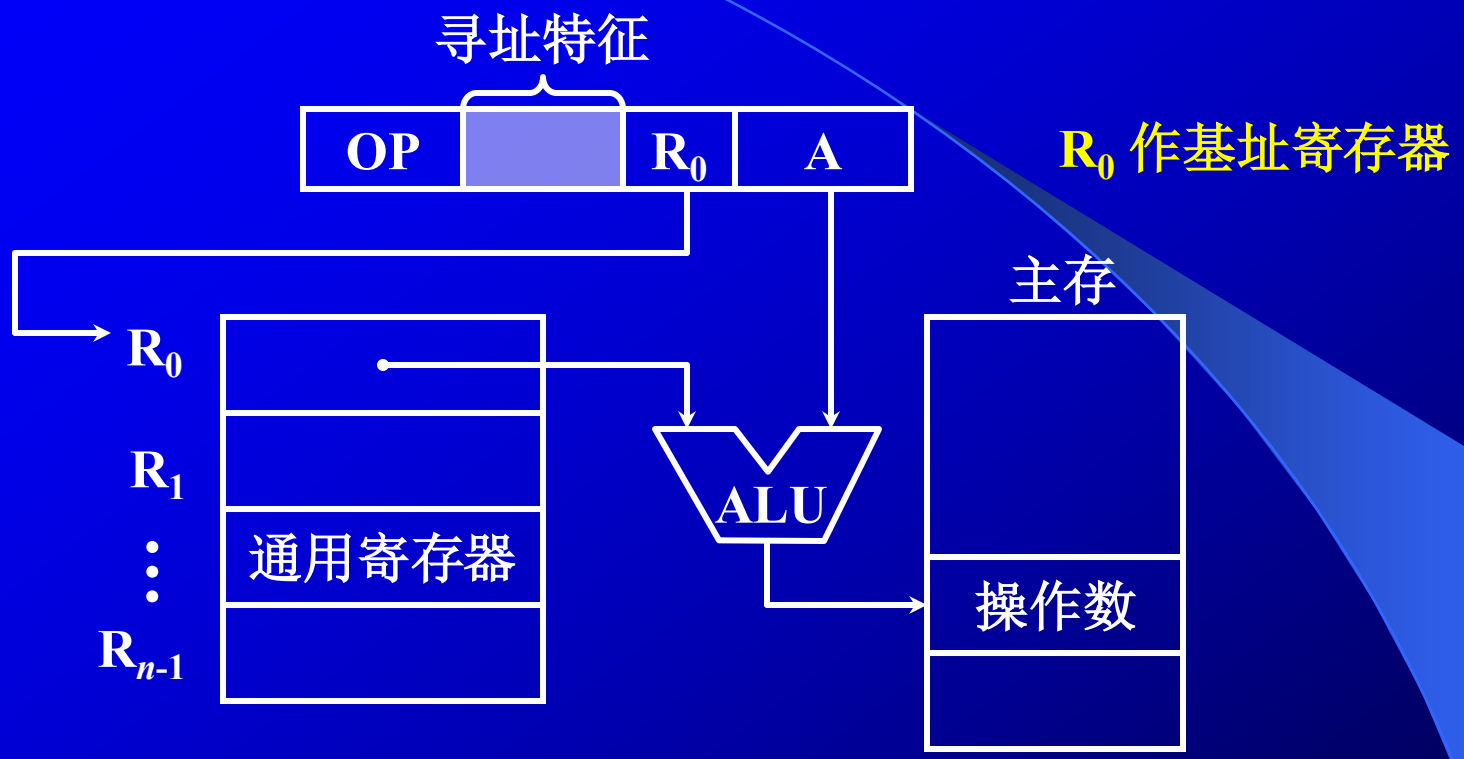
(1) 采用专用寄存器作基址寄存器

EA = (BR) + A　　BR 为基址寄存器



- 可扩大寻址范围
- 有利于多道程序
- **BR 内容由操作系统或管理程序确定**
- 在程序的执行过程中 **BR** 内容不变，形式地址 **A** 可变

(2) 采用通用寄存器作基址寄存器



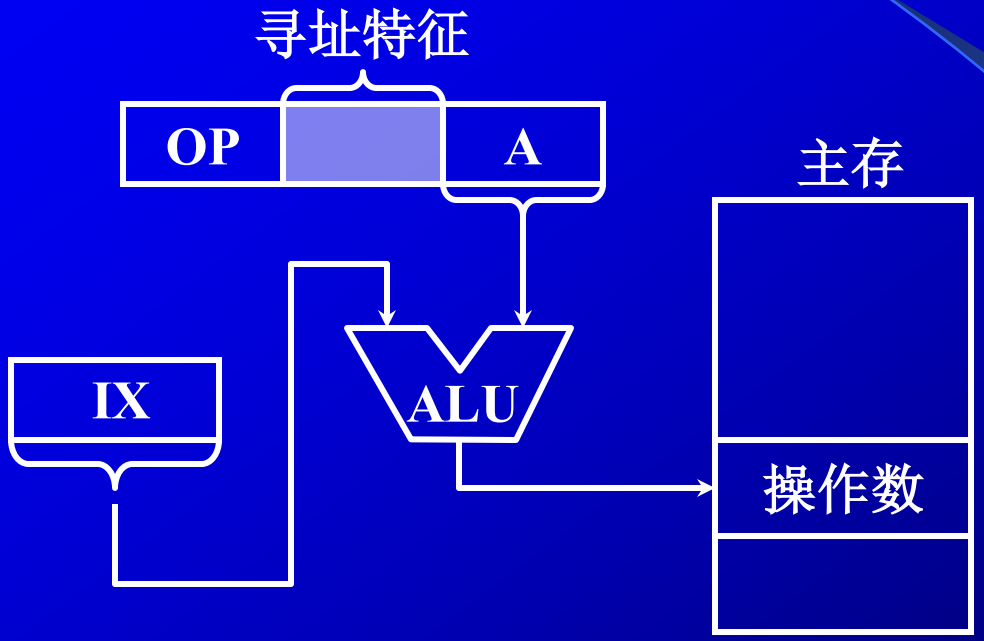
- 由用户指定哪个通用寄存器作为基址寄存器
- 基址寄存器的内容由操作系统确定
- 在程序的执行过程中 R_0 内容不变，形式地址 A 可变

8. 变址寻址

$EA = (IX) + A$

IX 为变址寄存器（专用）

通用寄存器也可以作为变址寄存器



- 可扩大寻址范围
- IX 的内容由用户给定
- 在程序的执行过程中 IX 内容可变，形式地址 A 不变
- 便于处理数组问题

例 设数据块首地址为 D，求 *N* 个数的平均值 7.3

直接寻址

```
LDA   D
ADD   D + 1
ADD   D + 2
  ⋮
ADD   D + ( N -1 )
DIV    # N
STA    ANS
```

共 *N* + 2 条指令

变址寻址

```
      LDA   # 0
      LDX   # 0
M     ADD   X, D
      INX
      CPX   # N
      BNE   M
      DIV   # N
      STA   ANS
```

X 为变址寄存器

D 为形式地址

(X) +1→ X

(X) 和 #*N* 比较

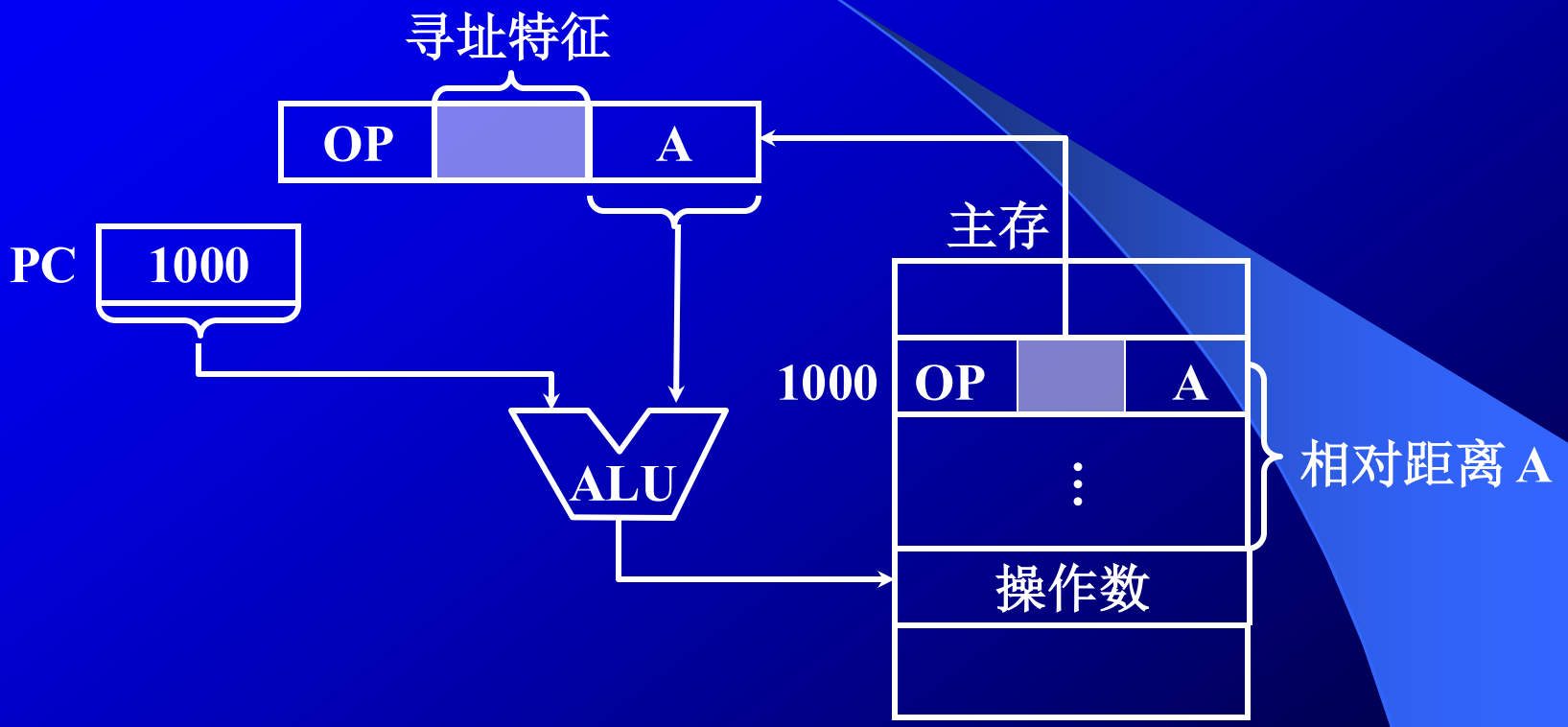
结果不为零则转

共 8 条指令

9. 相对寻址

$EA = (PC) + A$

A 是相对于当前指令的位移量（可正可负，补码）



- A 的位数决定操作数的寻址范围
- 程序浮动
- 广泛用于转移指令

(1) 相对寻址举例


```
        LDA    # 0
        LDX    # 0
M       ADD    X, D
M+1     INX
M+2     CPX    # N
M+3     BNE    M      ——→  * – 3
        DIV    # N
        STA    ANS
```

* 相对寻址特征

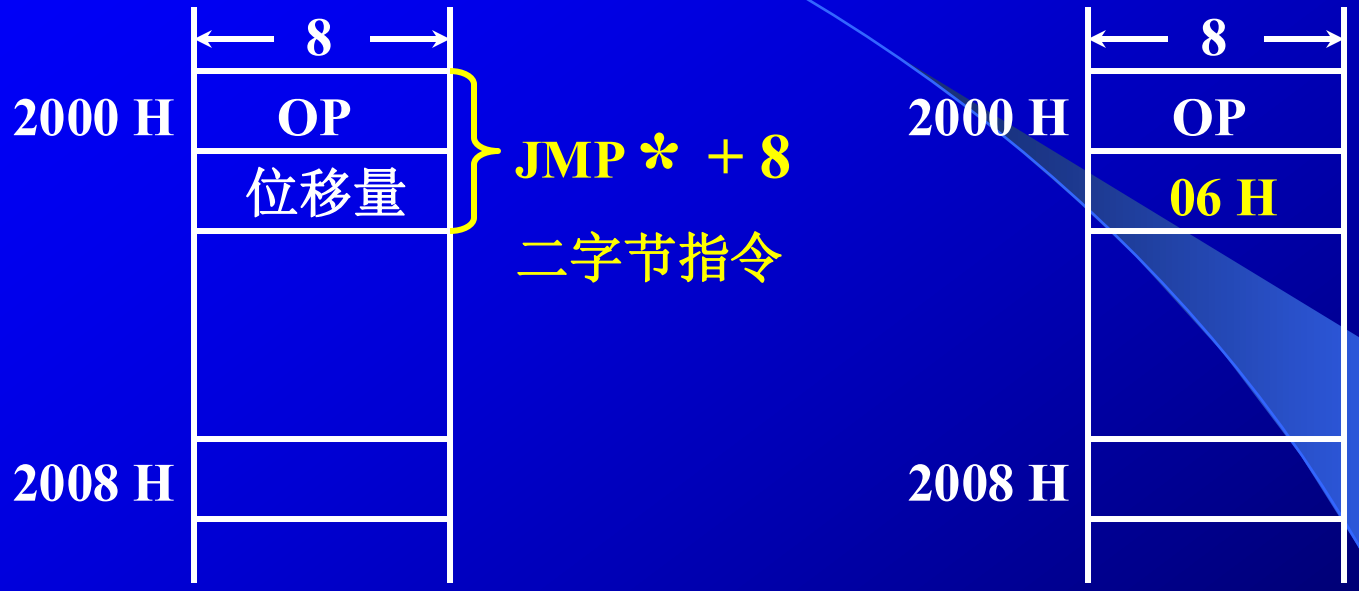
M 随程序所在存储空间的位置不同而不同

而指令 BNE * – 3 与 指令 ADD X, D 相对位移量不变

指令 BNE * – 3 操作数的有效地址为

EA = (M+3) – 3 = M

(2) 按字节寻址的相对寻址举例



设 当前指令地址 PC = 2000H

转移后的目的地址为 2008H

因为 取出 JMP * +8 后 PC = 2002H

故 JMP * +8 指令 的第二字节为 2008H - 2002H = 06H

10. 堆栈寻址

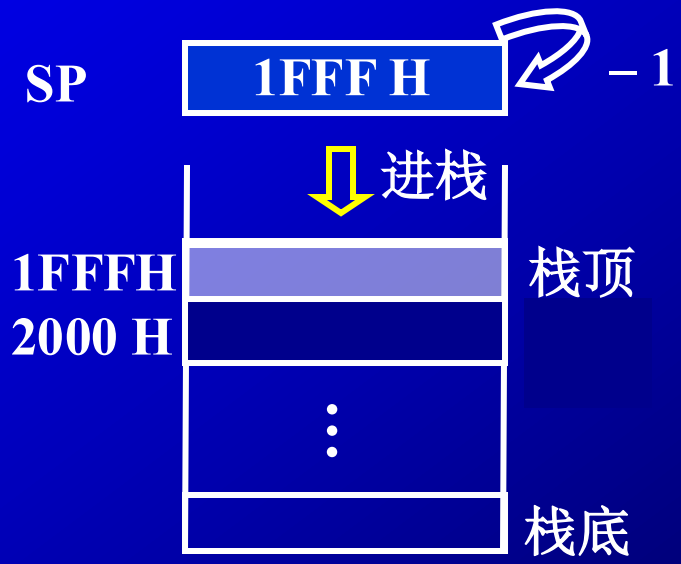
(1) 堆栈（Stack，栈）的特点

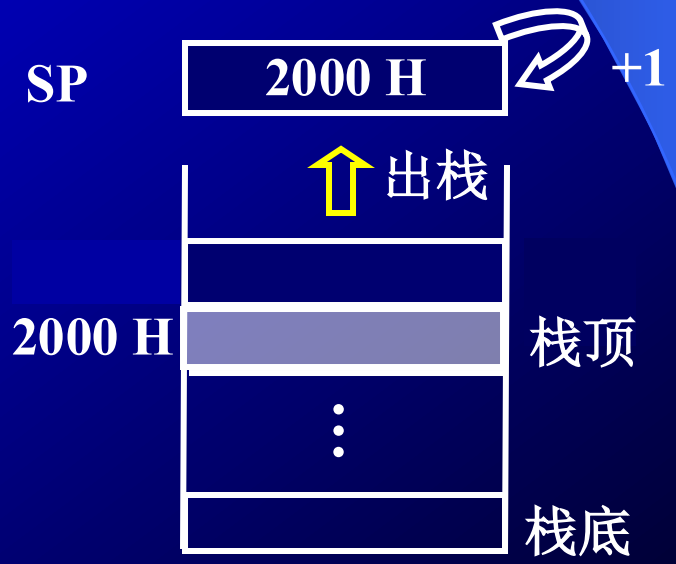
堆栈 { 硬堆栈　多个寄存器
　　 软堆栈　指定的存储空间

计算机中设置栈，或堆栈型指令系统

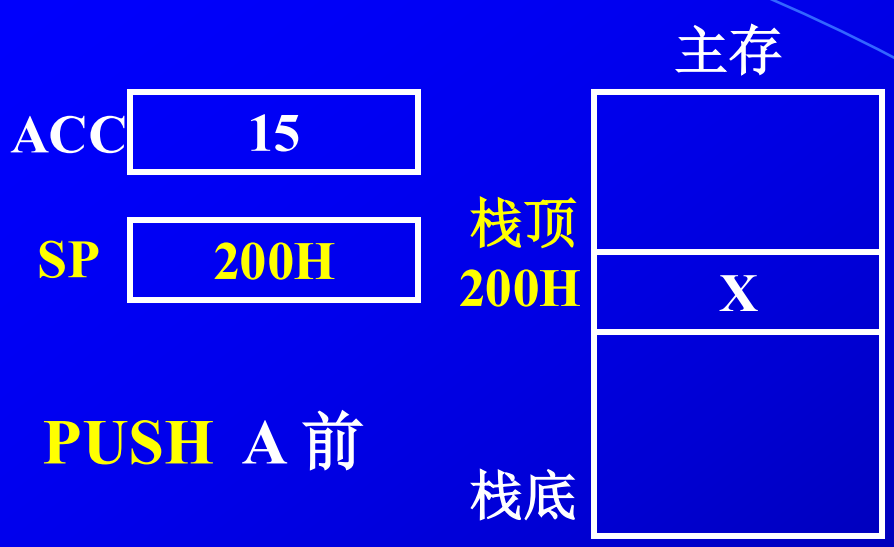
先进后出（一个入出口）　栈顶地址 由 SP 指出

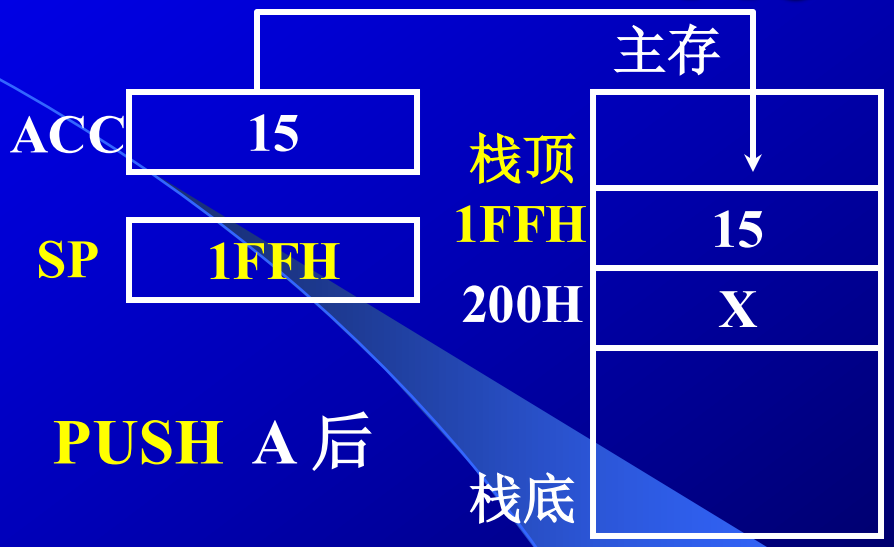
进栈　（SP）– 1 ⟶ SP　　出栈　（SP）+ 1 ⟶ SP

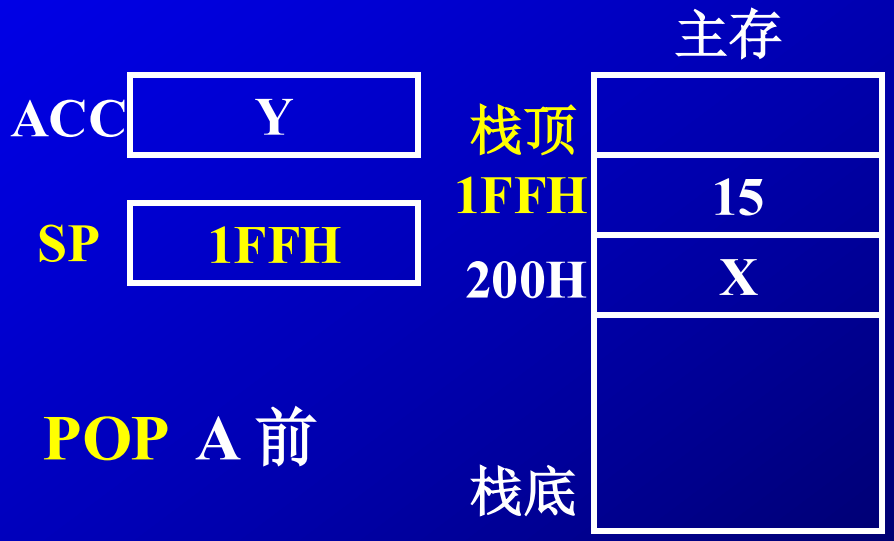


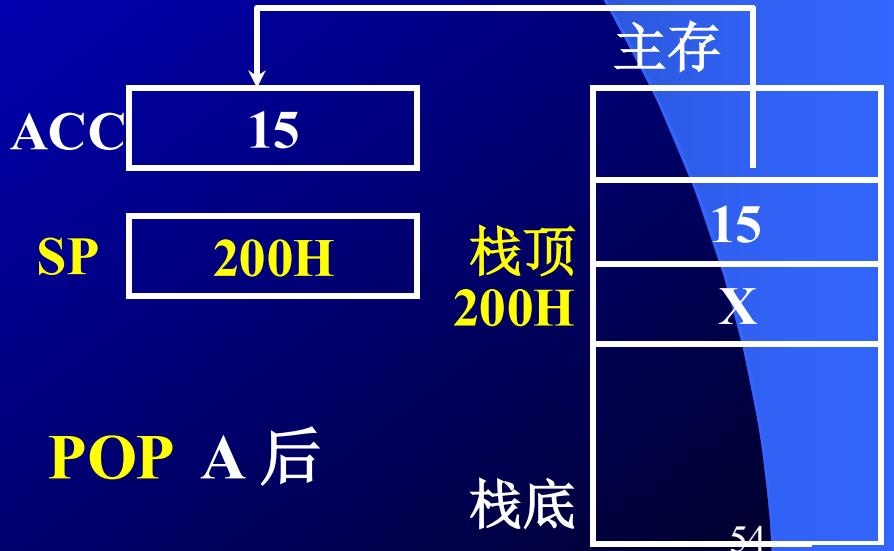


(2) 堆栈寻址举例


主存
ACC
15
SP
200H
栈顶
200H
X
栈底
PUSH A 前


主存
ACC
15
SP
1FFH
栈顶
1FFH
15
200H
X
栈底
PUSH A 后


主存
ACC
Y
SP
1FFH
栈顶
1FFH
15
200H
X
栈底
POP A 前


主存
ACC
15
SP
200H
15
栈顶
200H
X
栈底
POP A 后

(3) SP 的修改与主存编址方法有关

7.3

① 按 字 编址

进栈　（SP）– 1 ⟶ SP

出栈　（SP）+ 1 ⟶ SP

② 按 字节 编址

存储字长 16 位　进栈　（SP）– 2 ⟶ SP

出栈　（SP）+ 2 ⟶ SP

存储字长 32 位　进栈　（SP）– 4 ⟶ SP

出栈　（SP）+ 4 ⟶ SP

基本寻址方式的算法和优缺点

假设：A=地址字段值，R=寄存器编号，
EA=有效地址， (X)=X中的内容

方式	算法	主要优点	主要缺点
立即 #	操作数=A	指令执行速度快	操作数幅值有限
直接	EA=A	有效地址计算简单	地址范围有限
间接 @	EA=(A)	有效地址范围大	多次存储器访问
寄存器	操作数=(R)	指令执行快，指令短	地址范围有限
寄间接	EA=(R)	地址范围大	额外存储器访问
偏移	EA=(R)+A	灵活	复杂
堆栈	EA=栈顶	指令短	应用有限

偏移方式：将直接方式和寄存器间接方式结合起来
有：基址BR / 变址IX / 相对(*)PC 三种
MIPS不区分基址还是变址，统一为偏移寻址方式

补充问题：以上各种寻址方式下，操作数在寄存器中还是在存储器中？有没有可能存储在磁盘中？什么情况下，所取数据在磁盘中？

只有当操作数在存储器中时，才有可能“缺页”，此时操作数在磁盘中！

7.4 指令格式举例

一、设计指令格式时应考虑的各种因素

1. 指令系统的 兼容性 （向上兼容）

2. 其他因素

操作类型	包括指令个数及操作的难易程度
数据类型	确定哪些数据类型可参与操作
指令格式	指令字长是否固定
	操作码位数、是否采用扩展操作码技术，
	地址码位数、地址个数、寻址方式类型
寻址方式	指令寻址、操作数寻址
寄存器个数	寄存器的多少直接影响指令的执行时间

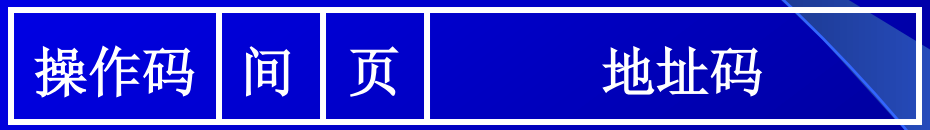
二、指令格式举例

7.4

1. PDP – 8

指令字长固定 12 位
只有1个通用寄存器ACC
累加器型指令系统

访存类指令



I/O 类指令

寄存器类指令

采用扩展操作码技术，共35条指令

2. PDP – 11

指令字长有 16 位、32 位、48 位三种

零地址 (16 位)

扩展操作码技术

一地址 (16 位)

二地址 R – R (16 位)

二地址 R – M (32 位)

二地址 M – M (48 位)

采用专门的寻址方式字段, S、D：3位
指定寻址方式，3位为寄存器编号

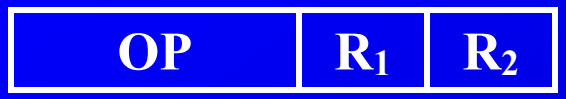
3. IBM 360

Ri:寄存器　X:变址器

Bi:基址器　Di:位移量

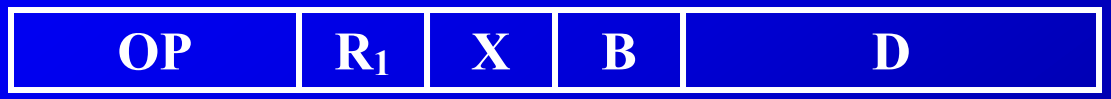
I:立即数　L:数的长度

RR 格式



二地址 R – R

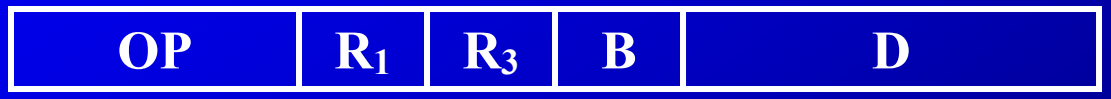
RX 格式



二地址 R – M

基址加变址寻址

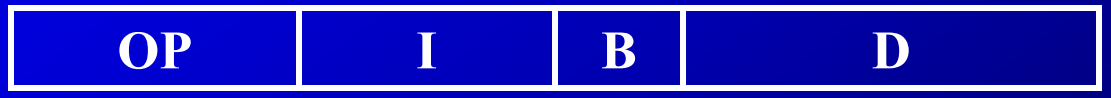
RS 格式



三地址 R – M

基址寻址

SI 格式



立即数 – M

基址寻址

SS 格式

二地址 M – M

基址寻址

RR:寄存器-寄存器　RX:寄存器-变址存储器

SI:基址存储器-立即数　SS:基址存储器-基址存储器

4. Intel 8086

7.4

(1) 指令字长　1 ~ 6 个字节

INC　AX　　1 字节

MOV　WORD PTR[0204], 0138H　　6 字节

(2) 地址格式

零地址　NOP　　1 字节

一地址　CALL　段间调用　　5 字节

CALL　段内调用　　3 字节

二地址　ADD AX，BX　　2 字节　寄存器 – 寄存器

ADD AX，3048H　　3 字节　寄存器 – 立即数

ADD AX，[3048H]　4 字节　寄存器 – 存储器

7.5 RISC 技术

三十年前的论战

“我们认为，基于RISC理念设计的处理器只有在极少数情况下慢于CISC处理器……过多的指令使得CISC处理器的控制逻辑复杂……研发成本上升……编译器也不知道该如何利用这么复杂的指令集……CISC的设计思路应当反思。” —— RISC的早期倡导者之一，David Patterson

“RISC与CISC的区别缺乏明确定义，而且RISC缺乏有力实验证明其宣称的优势，仅停留在纸面的设计是不够的，我们在VAX结构的设计中发现很多与RISC理念相反的地方……实验数据证明RISC的出发点有误……”—— CISC结构的设计者代表，Douglas W. Clark和William D. Strecker.

7.5

一、RISC 的产生和发展

RISC（Reduced Instruction Set Computer）

CISC（Complex Instruction Set Computer）

80 — 20 规律　　—— RISC技术

- 典型程序中 80% 的语句仅仅使用处理机中 20% 的指令
- 执行频度高的简单指令，因复杂指令的存在，执行速度无法提高

? 能否用 20% 的简单指令组合不常用的 80% 的指令功能

二、RISC 的主要特征

- 选用使用频度较高的一些 简单指令，
  复杂指令的功能由简单指令来组合
- 指令 长度固定、指令格式种类少、寻址方式少
- 只有 LOAD / STORE 指令访存
- CPU 中有多个 通用 寄存器
- 采用 流水技术 一个时钟周期 内完成一条指令
- 采用 组合逻辑 实现控制器
- 采用 优化 的 编译 程序

三、CISC 的主要特征

- 指令系统 复杂庞大，各种指令使用频度相差大
- 指令 长度不固定、指令格式种类多、寻址方式多
- 访存 指令 不受限制
- CPU 中设有 专用寄存器
- 大多数指令需要 多个时钟周期 执行完毕
- 采用 微程序 控制器
- 难以 用 优化编译 生成高效的目的代码

四、RISC和CISC 的比较

7.5

1. RISC更能 充分利用 VLSI 芯片的面积
2. RISC 更能 提高计算机运算速度

   指令数、指令格式、寻址方式少，
   通用 寄存器多，采用 组合逻辑，
   便于实现 指令流水
3. RISC 便于设计，可 降低成本，提高 可靠性
4. RISC 有利于编译程序代码优化
5. RISC 不易 实现 指令系统兼容

作业

- 习题：7.12，7.13，7.15，7.16