计算机组成原理

第七章 指令系统

阙夏
计算机与信息学院

2023/5/7

大　纲

四、指令系统

(一)指令系统的基本概念

(二)指令格式

(三)寻址方式

（四）数据的对齐和大/小端存放方式

（五）CISC 和 RISC 的基本概念

（六）高级语言程序与机器级代码之间的对应

1.编译器，汇编器和链路器的基本概念

2.选择结构语句的机器级表示

3.循环结构语句的机器级表示

4.过程（函数）调用对应的机器级表示

Contents

一、指令格式

指令：就是让计算机识别并执行某种操作的命令。

指令系统：一台计算机中所有机器指令的集合。
称为这台计算机的指令系统。

系列计算机：指基本指令系统相同且基本体系，结构相同的一系列计算机。

系列机能解决软件兼容问题的必要条件是该系列的各种机种有共同的指令集，而且新推出的机种的指令系统一定包含旧机种的所有指令，因此在旧机种上运行的各种软件可以不加任何修改地在新机种上运行。

https://baijiahao.baidu.com/s?id=1718417865583947761&wfr=spider&for=pc

7.1 机器指令

指令的一般格式：

n位操作码字段的指令系统最多能够表示2^n条指令。

操作码字段	地址码字段

1. 操作码　指令应该执行什么性质的操作和具有何种功能。反映机器做什么操作。

(1) **长度固定**（译码简单）

用于指令字长较长的情况，RISC

如 IBM 370　　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

(3) 扩展操作码技术（教材P301）

操作码的位数随地址数的减少而增加

OP	A_1	A_2	A_3	
0000	A_1	A_2	A_3	4 位操作码
0001	A_1	A_2	A_3	最多15条三地址指令
⋮	⋮	⋮	⋮	
1110	A_1	A_2	A_3	
1111	0000	A_2	A_3	8 位操作码
1111	0001	A_2	A_3	最多15条二地址指令
⋮	⋮	⋮	⋮	
1111	1110	A_2	A_3	
1111	1111	0000	A_3	12 位操作码
1111	1111	0001	A_3	最多15条一地址指令
⋮	⋮	⋮	⋮	
1111	1111	1110	A_3	
1111	1111	1111	0000	16 位操作码
1111	1111	1111	0001	16条零地址指令
⋮	⋮	⋮	⋮	
1111	1111	1111	1111	

(3)扩展操作码技术

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

8 位操作码

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

12 位操作码

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

16 位操作码

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

两个原则：
1.编码不能重复
2.短码不能为长码前缀

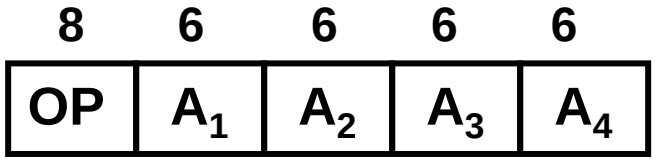
三地址指令操作码
每减少一种最多可多构成
2^4 种二地址指令

二地址指令操作码
每减少一种最多可多构成
2^4 种一地址指令

使用频率高用短码
使用频率低用长码

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

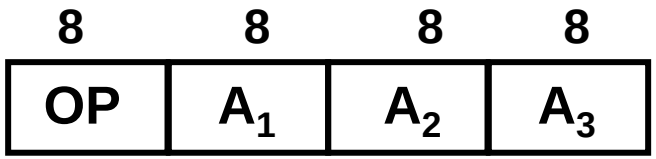
操作码固定为 8 位

4 次访存

寻址范围 $2^6 = 64$

若 PC 代替 A_4

(2) 三地址



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

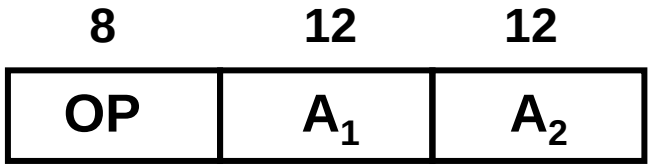
4 次访存

寻址范围 $2^8 = 256$

若 A_3 用 A_1 或 A_2 代替

7.1 机器指令

(3) 二地址



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

或

(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

1.无需任何操作数，如空操作指令、停机指令等；2.所需操作数地址是默认的。

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 = 机器字长（早期计算机）

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

单选题 1分

设置

【2017统考真题】某计算机按字节编址，指令字长固定且只有两种指令格式，其中三地址指令29条、二地址指令107 条，每个地址字段为6位，则指令字长至少应该是( )。

A 24位

B 26位

C 28位

D 32位

提交

7.2 操作数类型和操作种类

一、操作数类型

地址	绝对地址：无符号整数；相对地址：有符号数
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

例7.1 12345678H，分别用大端小端方式存储

存储器中的数据存放：边界对齐

大端

高位字节 地址为字地址

小端

低位字节 地址为字地址

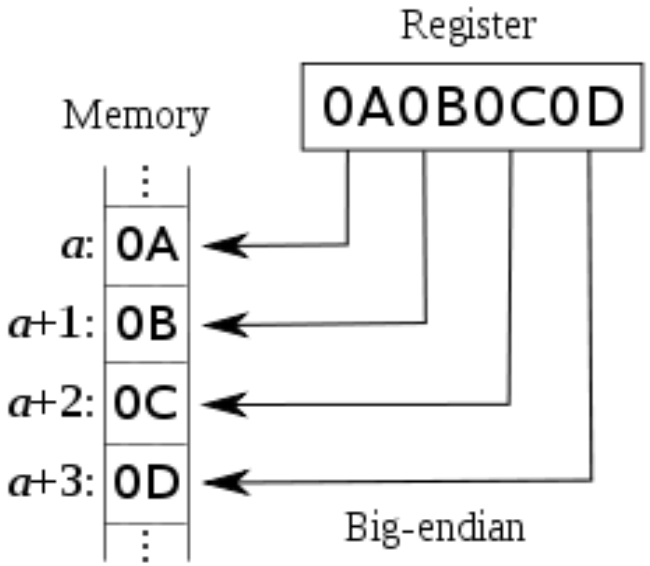
BIG Endian versus Little Endian

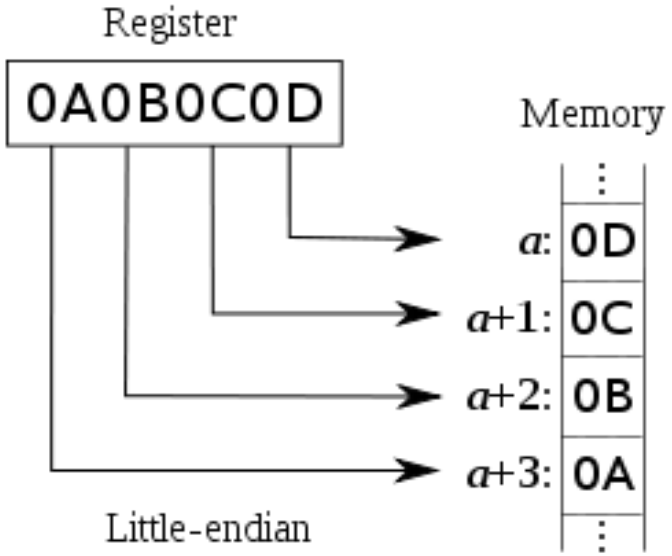
大端Big-Endian

低地址存放最高有效位（**MSB**），既高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

小端Little-Endian

低地址存放最低有效位（LSB），既低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。





7.2 操作数类型和操作种类

三、操作类型

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、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

AAA AAS AAM AAD

AND OR NOT XOR TEST

7.2 操作数类型和操作种类

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

(2) 条件转移

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

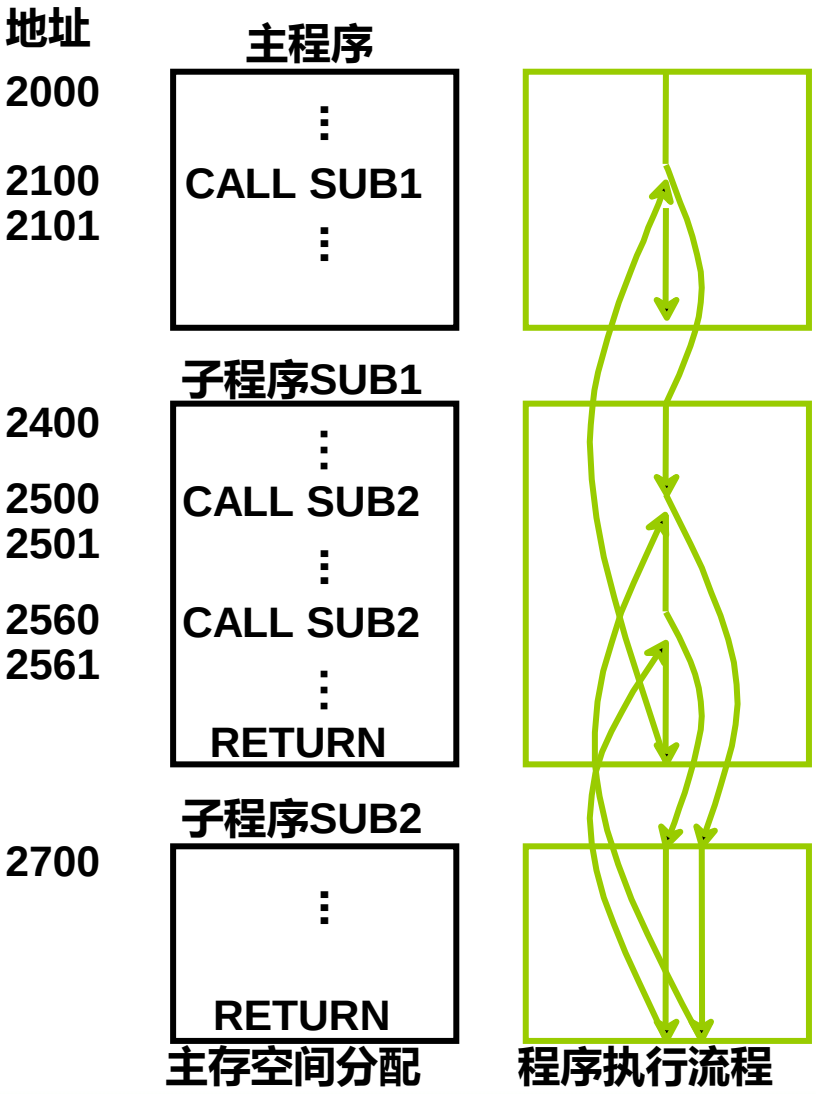
```
300
 ⋮
305
306
307
```

完成触发器

SKP DZ D = 0 则跳

7.2 操作数类型和操作种类

(3) 调用和返回



7.2 操作数类型和操作种类

(4) 陷阱（Trap）与陷阱指令

陷阱：计算机系统的意外事故（如电压不稳。故障等）。
陷阱指令：是处理陷阱的指令。

- 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　AL, *m*　　IN　AL, DX

出　　CPU 的寄存器 ⟶ 端口地址

如　　OUT　*n*, AL　　OUT　DX, AL

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**

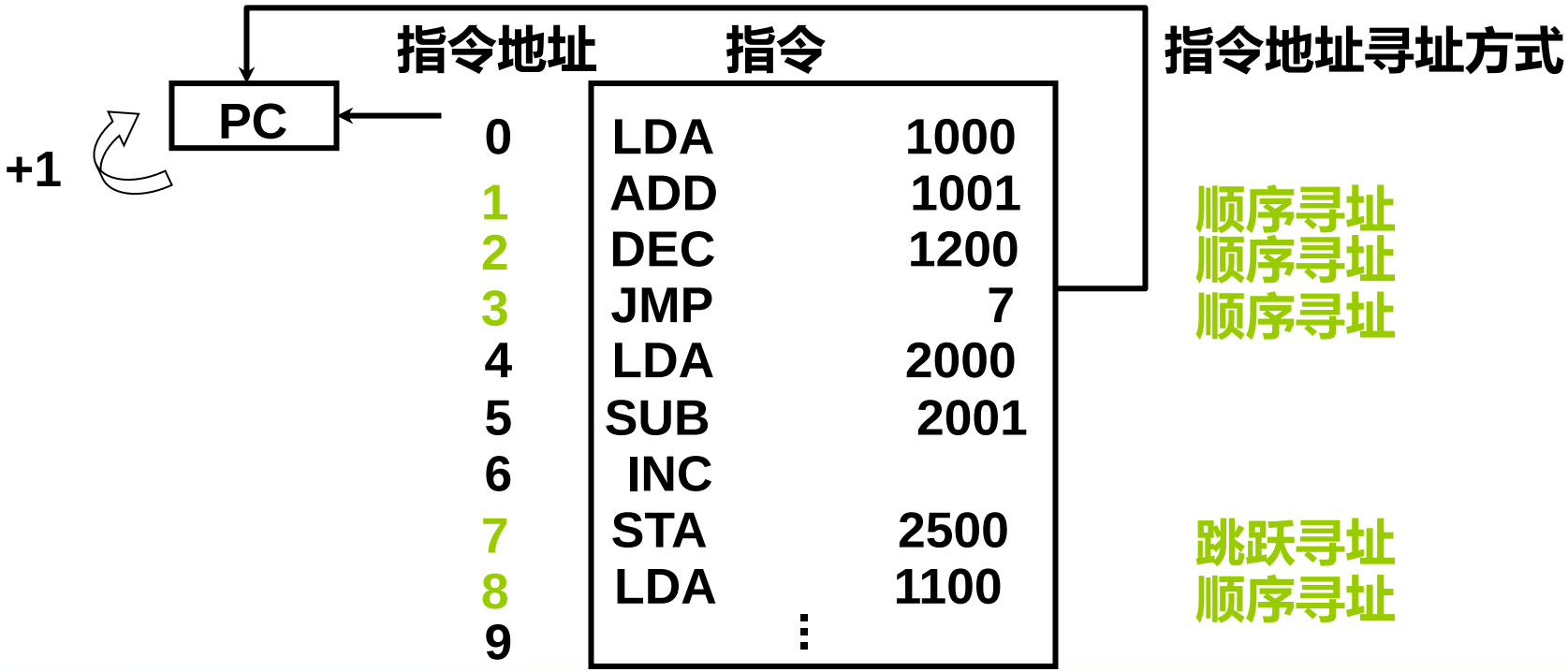
寻址方式
- 指令寻址
- 数据寻址

7.3 寻址方式

一、指令寻址

这里的+1是指下一条指令

- 顺序　　(PC) + 1 ⟶ PC
- 跳跃　　由转移指令指出，如 JMP



7.3 寻址方式

二、数据寻址

操作码	寻址特征	形式地址 A

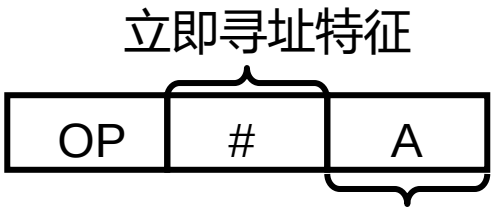
形式地址 指令字中的地址 **A**

有效地址 操作数的真实地址 **EA**

约定 **指令字长 = 存储字长 = 机器字长**

1. 立即寻址

形式地址 A 就是操作数



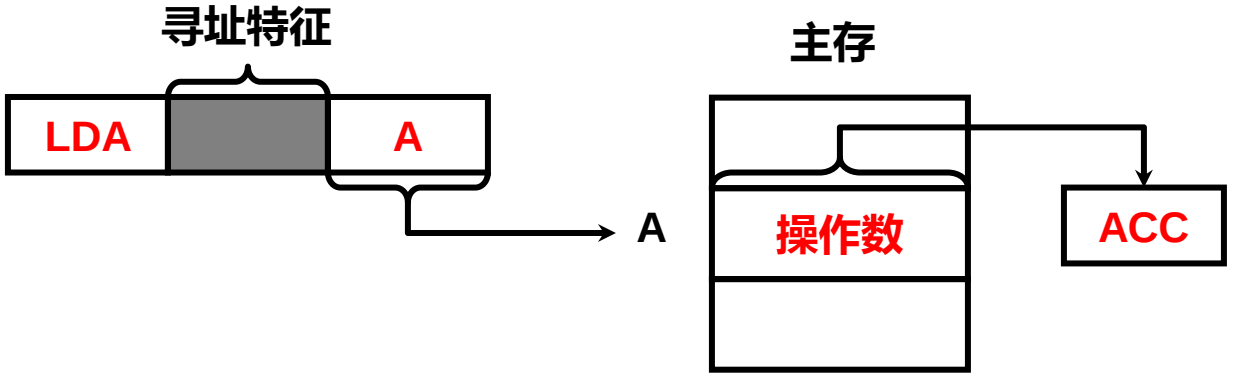
可正可负 **补码**

- **指令执行阶段不访存**
- **A 的位数限制了立即数的范围**

7.3 寻址方式

2. 直接寻址

EA = A　　有效地址由形式地址直接给出

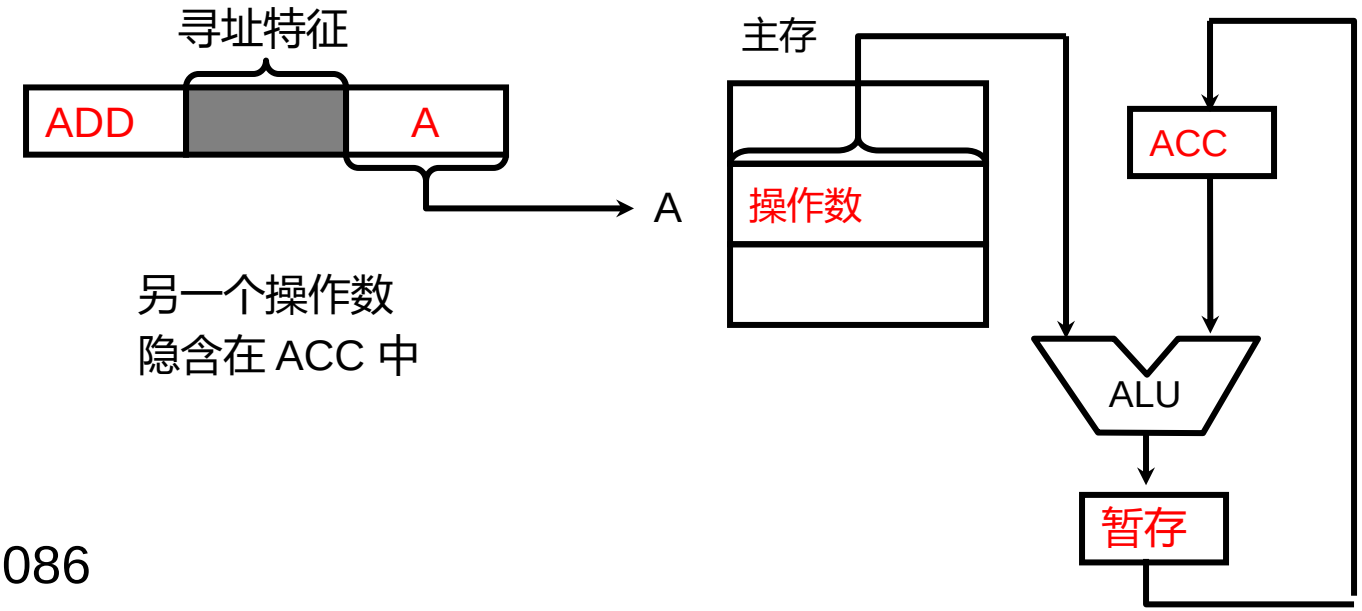


- 执行阶段访问一次存储器
- A 的位数决定了该指令操作数的寻址范围
- 操作数的地址不易修改（必须修改A）

7.3 寻址方式

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

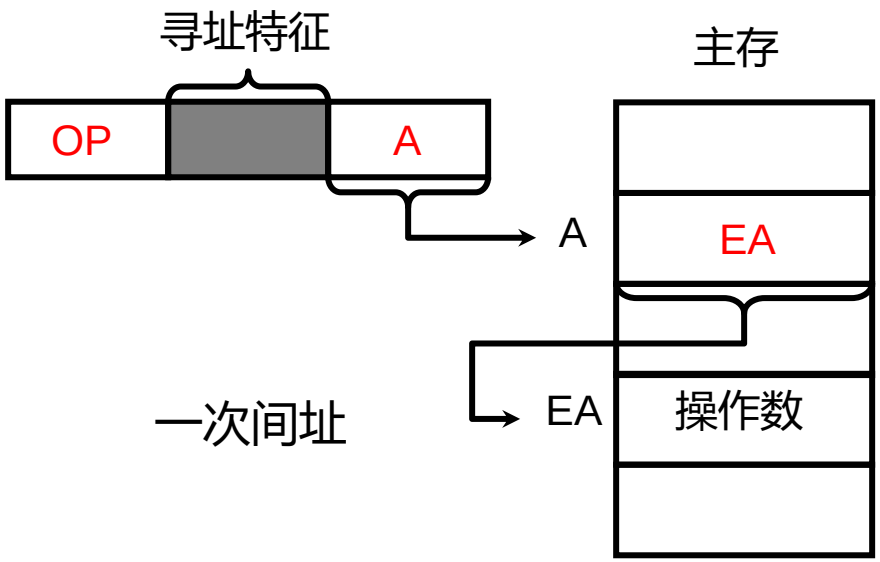
目的操作数的地址隐含在 DI 中

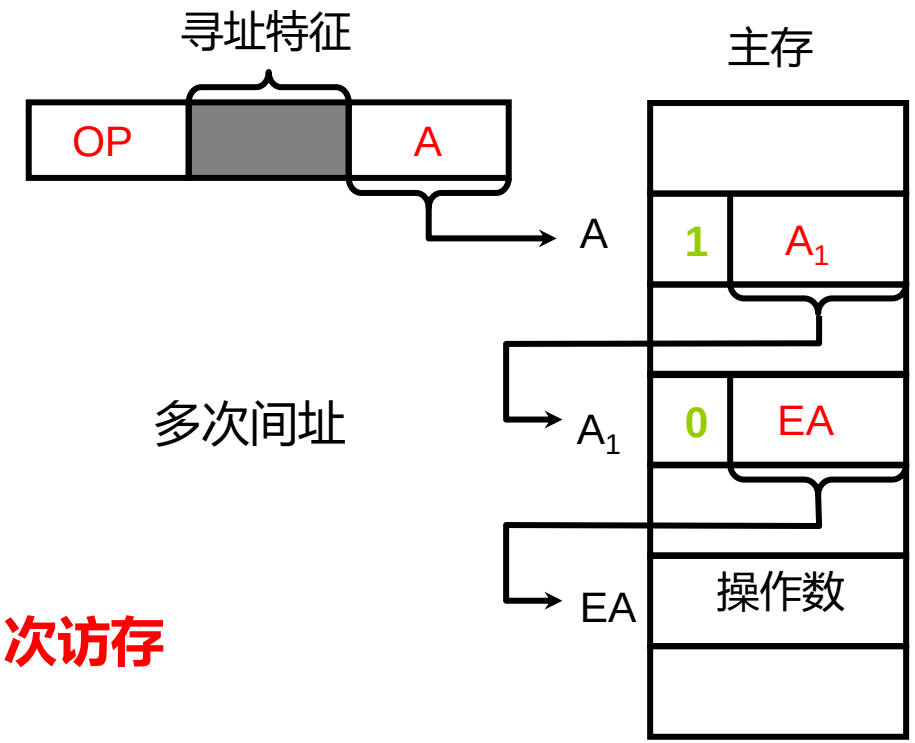
- **指令字中少了一个地址字段，可缩短指令字长**

7.3 寻址方式

4. 间接寻址

EA = （A）　有效地址由形式地址间接提供



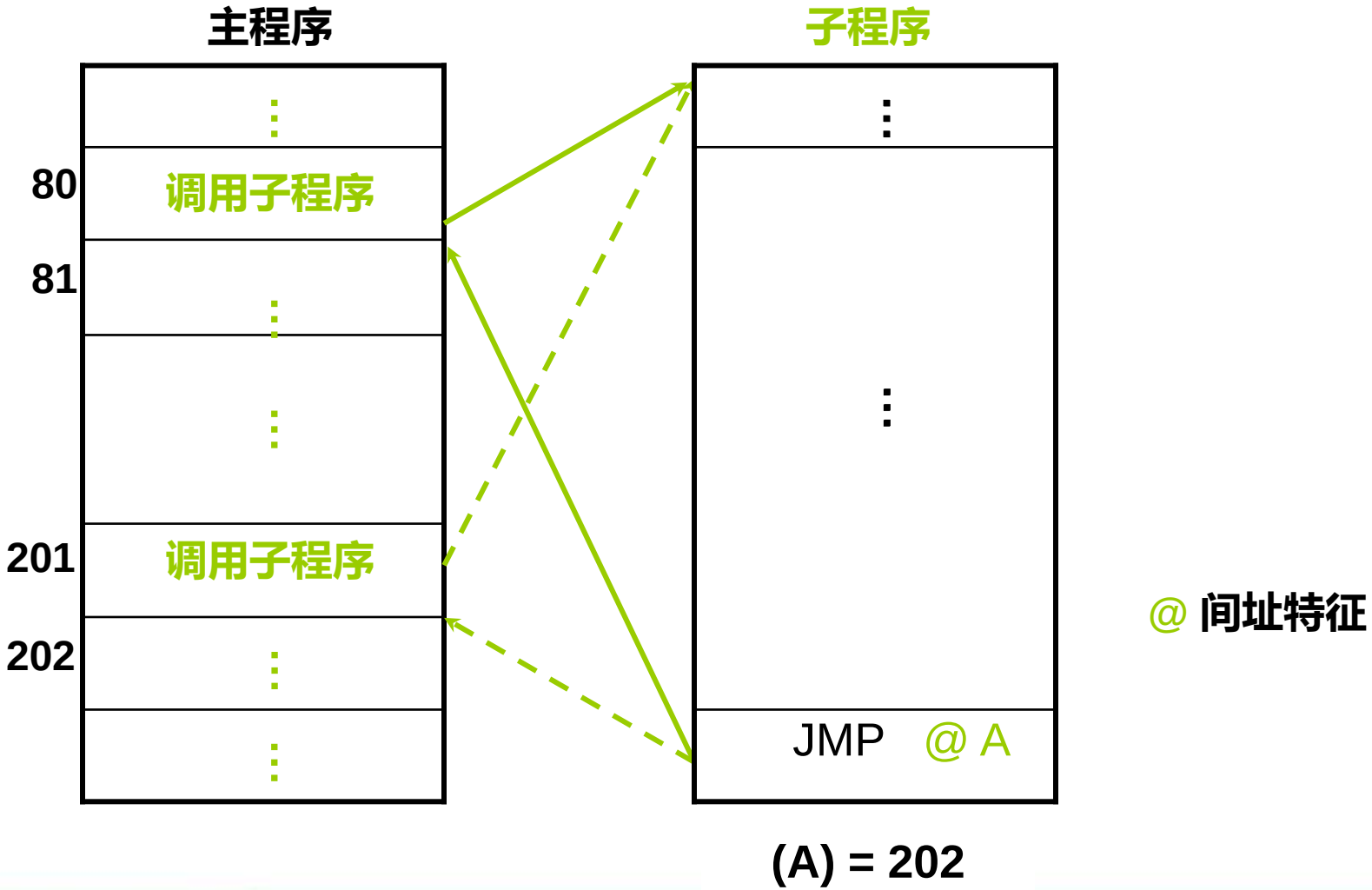


- 执行指令阶段 2 次访存
- 可扩大寻址范围
- 便于编制程序

多次访存

7.3 寻址方式

间接寻址编程举例

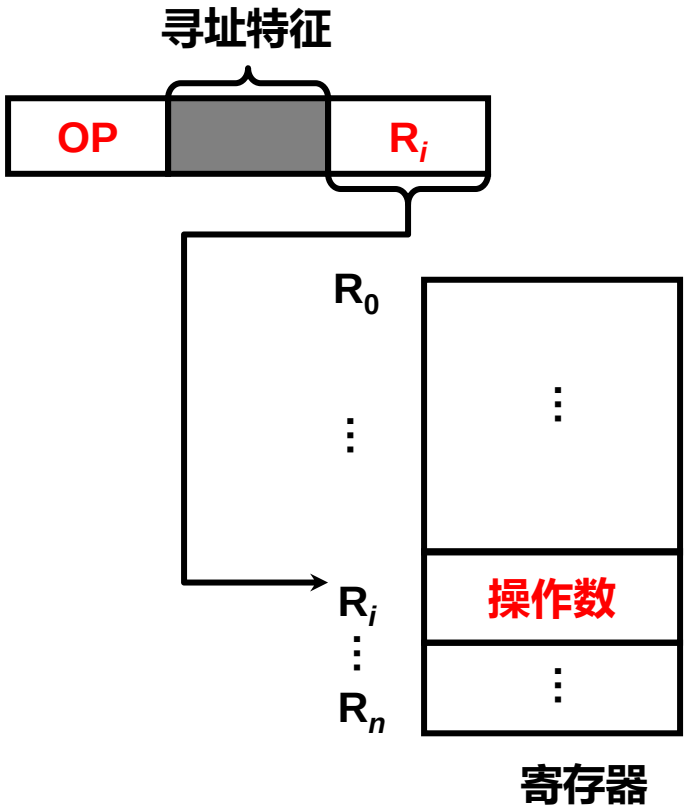


7.3 寻址方式

5. 寄存器（直接）寻址

EA = R_i

有效地址即为寄存器编号

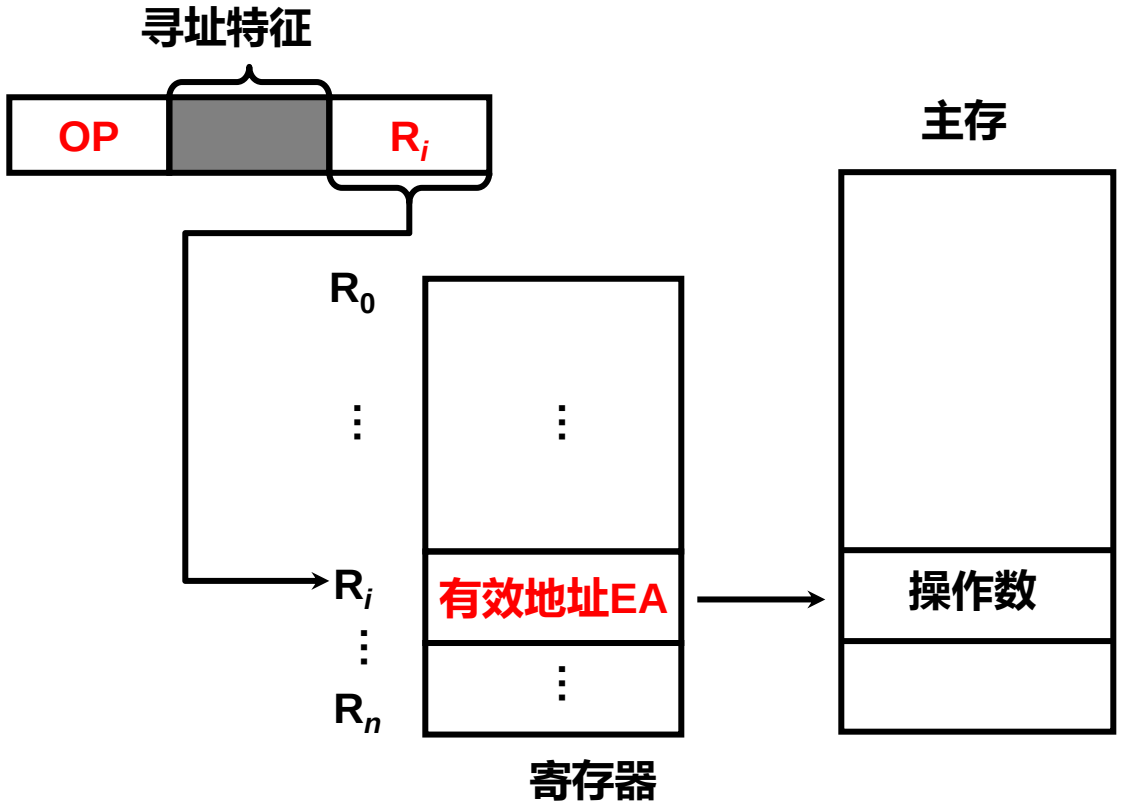


- 执行阶段不访存，只访问寄存器，执行速度快
- 寄存器个数有限，可缩短指令字长

7.3 寻址方式

6. 寄存器间接寻址

EA = (R_i)　　有效地址在寄存器中

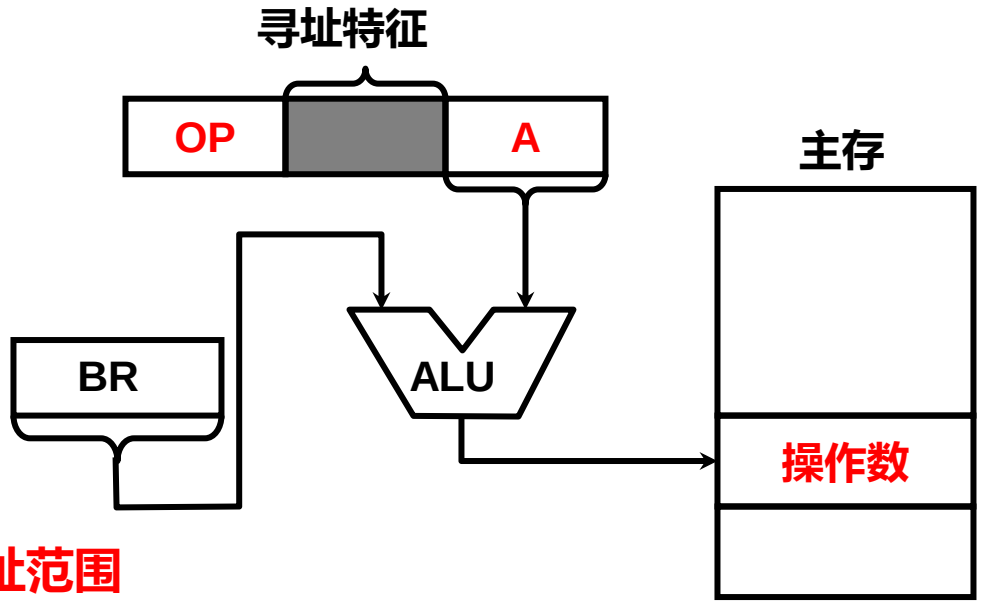


- 有效地址在寄存器中， 操作数在存储器中，执行阶段访存
- 比一般间接寻址速度快

7.3 寻址方式

7. 基址寻址 (1) 采用专用寄存器作基址寄存器

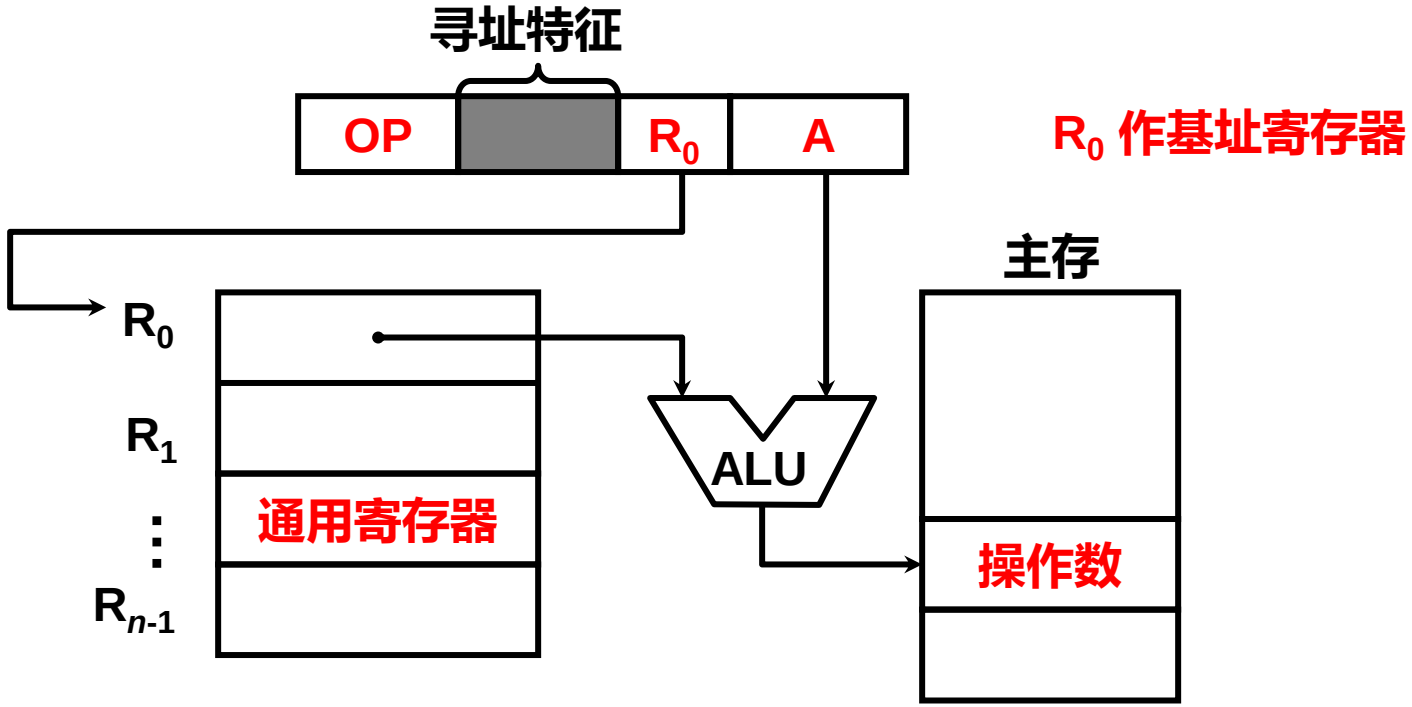
EA = (BR) + A　　BR 为基址寄存器



- 可扩大寻址范围
- 有利于多道/浮动程序，但偏移量偏小
- 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

7.3 寻址方式

(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

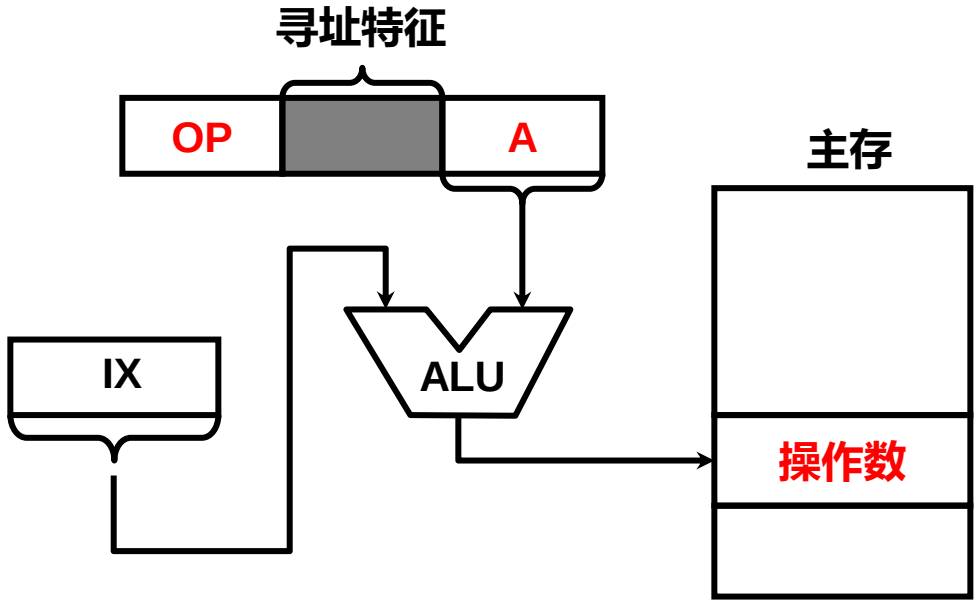
7.3 寻址方式

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
- 便于处理循环/数组问题

例7.2 设数据块首地址为 D，求 *N* 个数的平均值

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          0 ⟶ ACC
      LDX   # 0          X 为变址寄存器
M     ADD   X, [D]       D 为形式地址
      INX                (X) +1 ⟶ X
      CPX   # N          (X) 和 #N 比较
      BNE   M            结果不为零则转
      DIV   # N
      STA   ANS
```

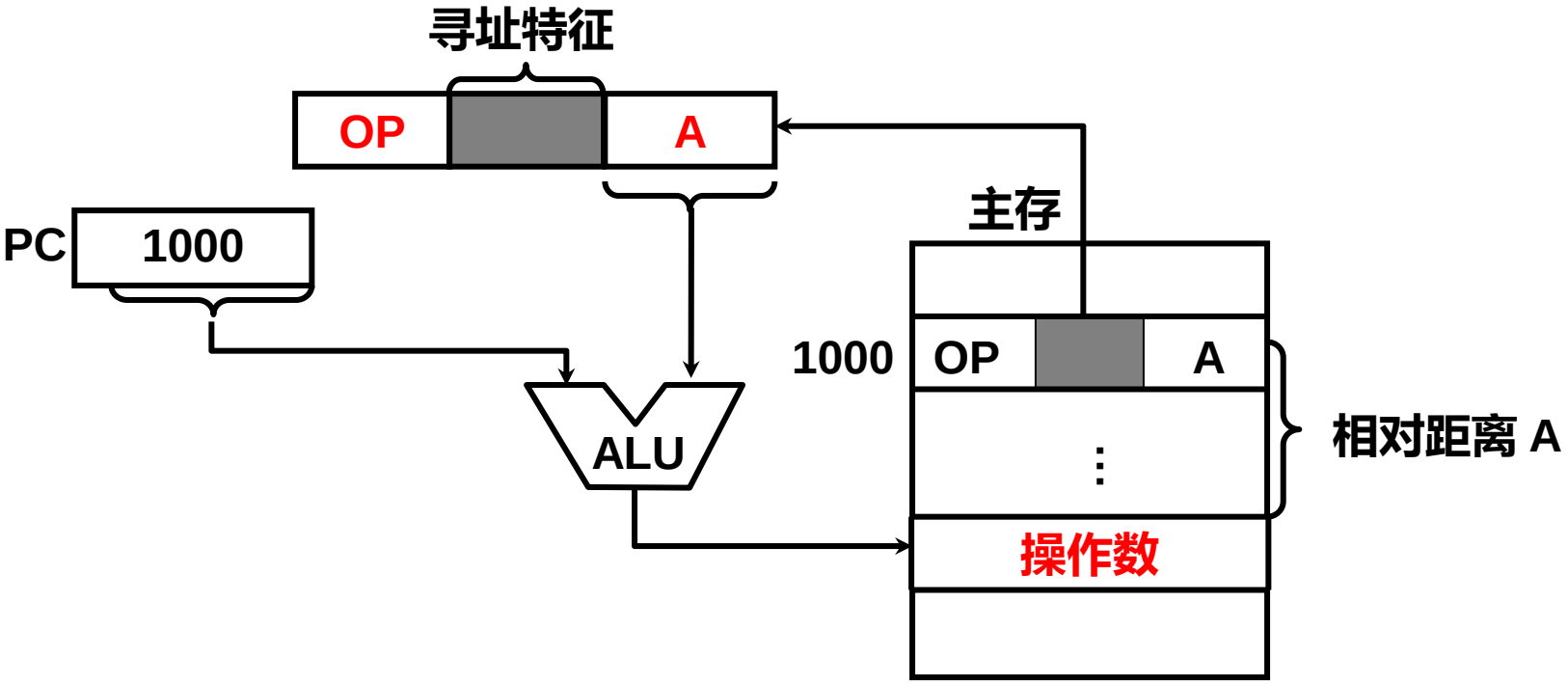
共 8 条指令

7.3 寻址方式

9. 相对寻址

EA = (PC) + A

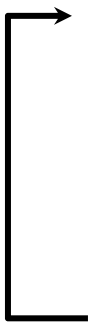
A 是相对于当前指令的位移量（可正可负，补码）



- A 的位数决定操作数的寻址范围
- 可用于编制浮动程序
- 广泛用于转移指令

7.3 寻址方式

(1) 相对寻址举例



	LDA	# 0	
	LDX	# 0	
M	ADD	X, D	
M+1	INX		
M+2	CPX	# N	* 相对寻址特征
M+3	BNE	M	* – 3
	DIV	# N	
	STA	ANS	

M 随程序所在存储空间的位置不同而不同

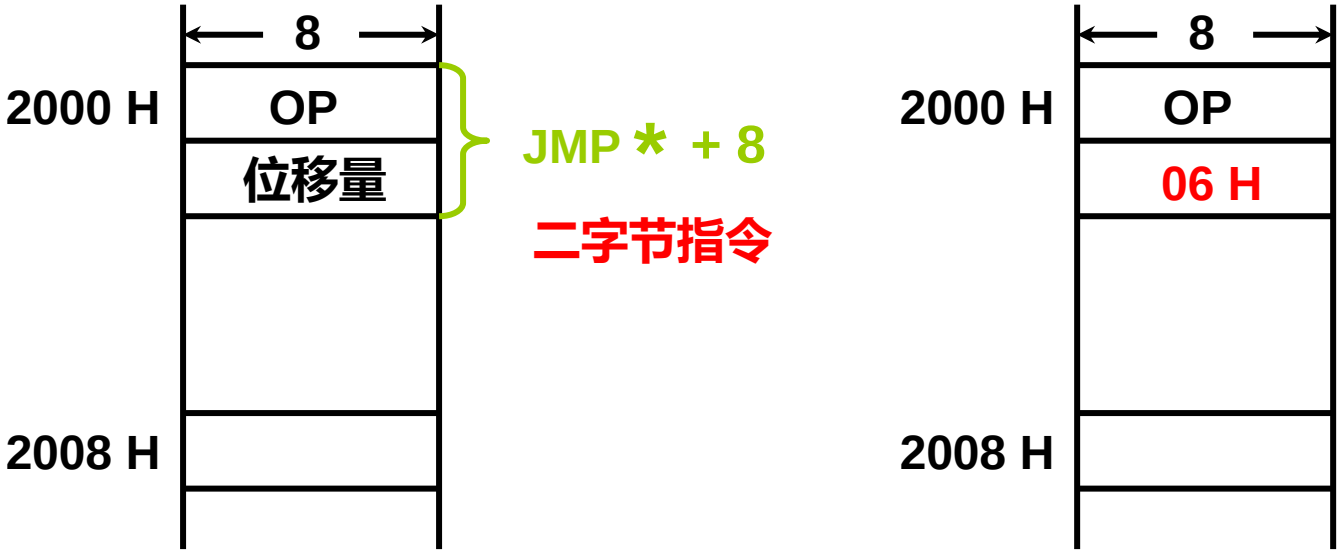
而指令 BNE * – 3 与 指令 ADD X, D 相对位移量不变

指令 BNE * – 3 操作数的有效地址为

EA = (M+3) – 3 = M

7.3 寻址方式

(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

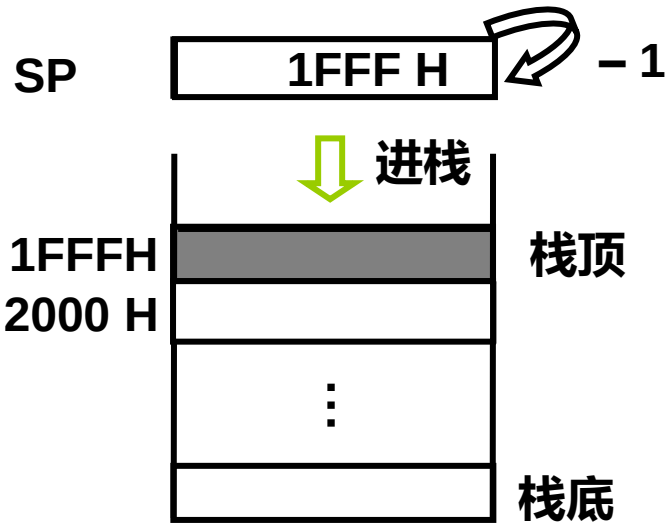
10. 堆栈寻址

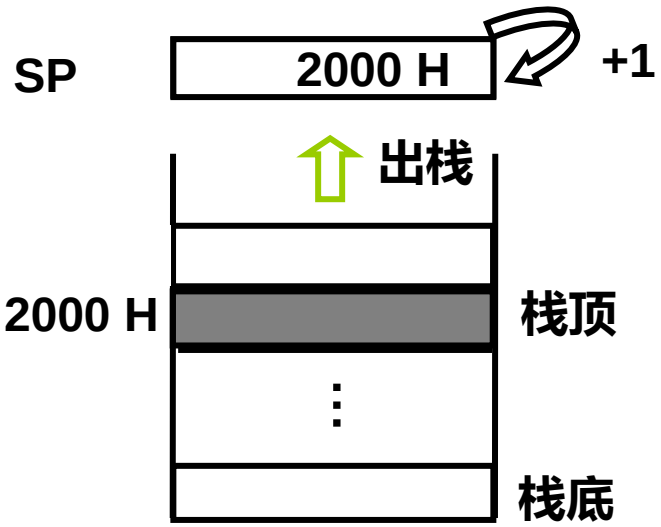
(1) 堆栈的特点

堆栈 { 硬堆栈 多个寄存器 堆栈计算机
软堆栈 指定的存储空间 8086计算机

先进后出（一个入出口） **栈顶地址** 由 SP 指出

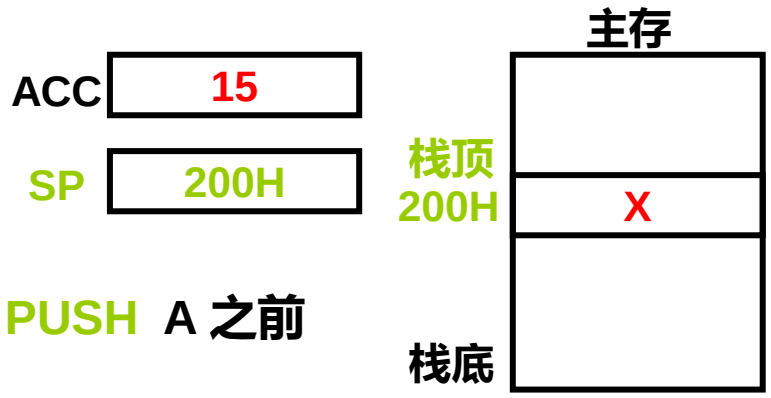
进栈 (SP) – 1 ⟶SP **出栈** (SP) + 1 ⟶ SP

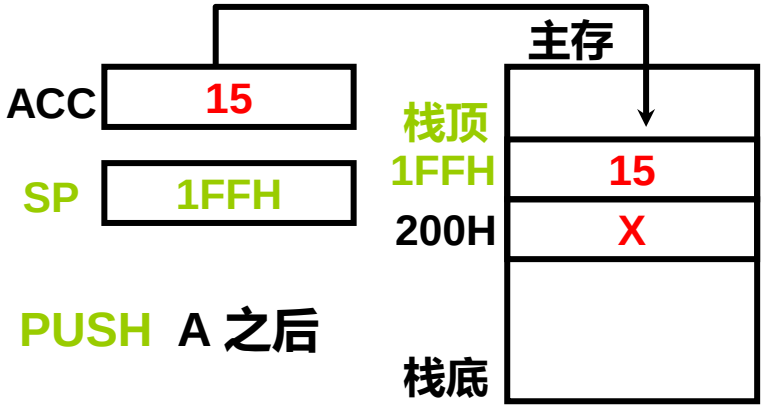


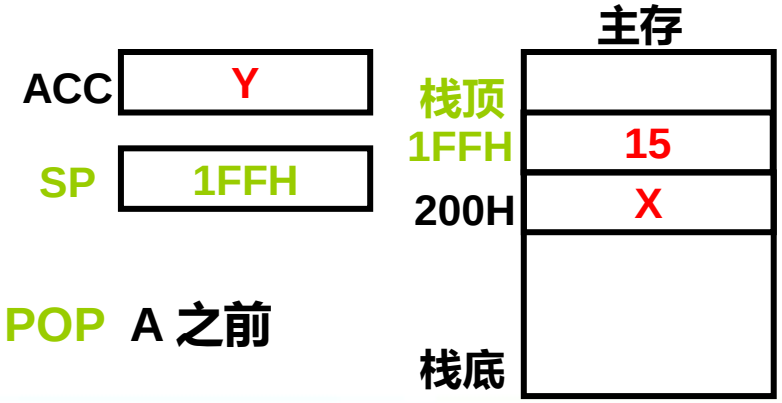


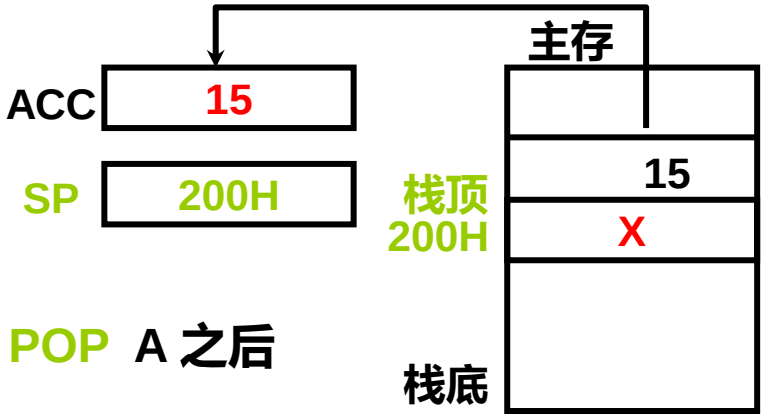
10. 堆栈寻址

(2) 堆栈寻址举例









7.3 寻址方式

(3) SP 的修改与主存编址方法有关

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

单选题 1分

设置

【2017统考真题】下列寻址方式中，最适合按下标顺序访问一维数组元素的是( ) 。

A 相对寻址

B 寄存器寻址

C 直接寻址

D 变址寻址

提交

填空题 2分

设置

假设（R）=1000，（1000）=2000，（2000）=3000，（3000）=5000，（PC）=4000，则以下寻址方式下访问到的指令操作数的值是多少?

(1)直接寻址，操作数值 [填空1]

(2)间接寻址，操作数值 [填空2]

(3) 寄存器间接寻址，操作数值 [填空3]

(4) 相对寻址，操作数值 [填空4]

作答

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）　　教材P320

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

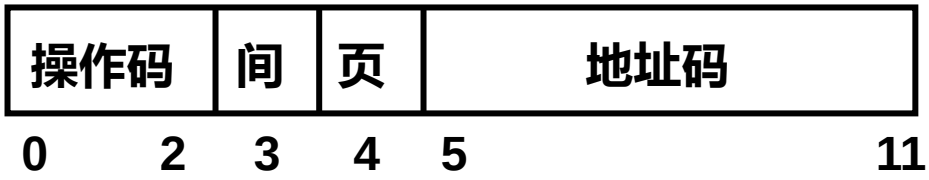
二、指令格式举例

1. PDP – 8

指令字长固定 12 位

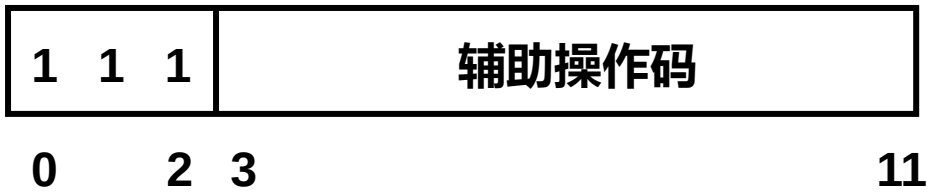
教材P320-321

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

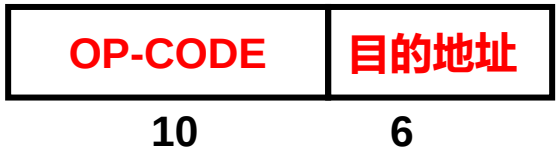
2. PDP – 11

指令字长有 16 位、32 位、48 位三种

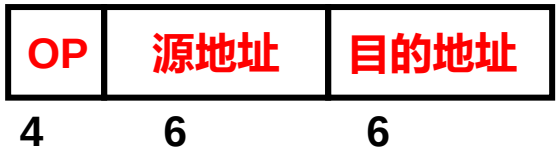
教材P321

零地址 (16 位)

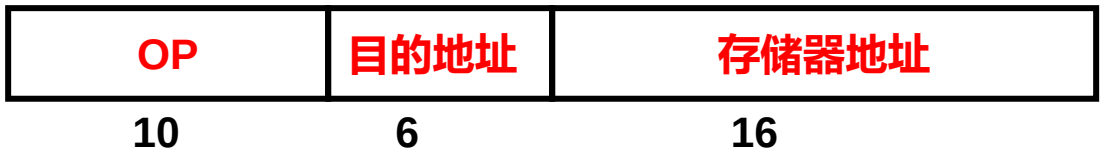
扩展操作码技术



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)

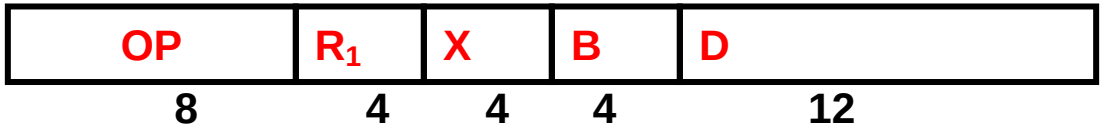
3. IBM 360

教材P322

RR 格式

二地址 R – R

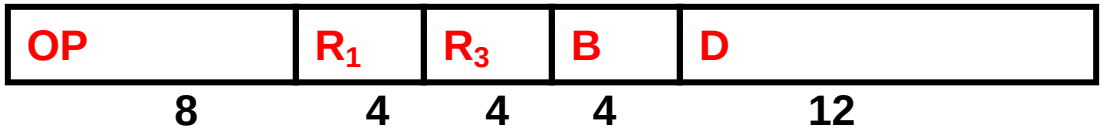
RX 格式



二地址 R – M

基址加变址寻址

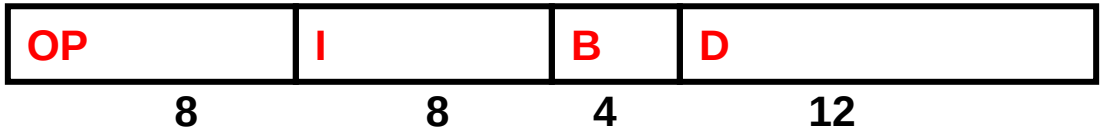
RS 格式



三地址 R – M

基址寻址

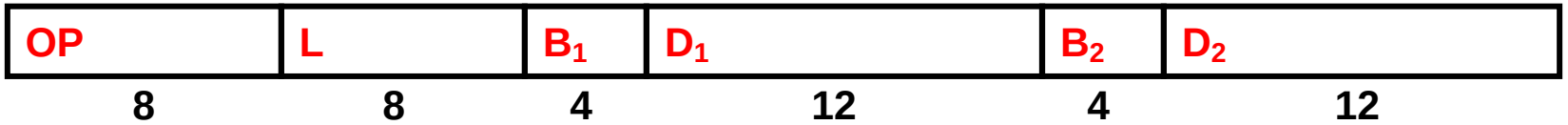
SI 格式



立即数 – M

基址寻址

SS 格式



二地址 M – M

基址寻址

(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

三、指令格式设计举例-自学* 7.4

教材P323-326

课后练习 7.4

7.19 CPU 内有 32 个 32 位的通用寄存器，设计一种能容纳 64 种操作的指令系统。假设指令字长等于机器字长，试回答以下问题。

(1) 如果主存可直接或间接寻址，采用寄存器－存储器型指令，能直接寻址的最大存储空间是多少？画出指令格式并说明各字段的含义。

(2) 在满足(1)的前提下，如果采用通用寄存器作基址寄存器，则上述寄存器－存储器型指令的指令格式有何特点？画出指令格式并指出这类指令可访问多大的存储空间？

中国慕课大学
刘宏伟

7.5 RISC 技术

一、RISC 的产生和发展

RISC（Reduced Instruction Set Computer）

CISC（Complex Instruction Set Computer）

20—80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

复杂指令集计算机CISC

- **CISC的缺陷**
  - 日趋庞大的指令系统不但使计算机的研制周期变长，而且难以保证设计的正确性，难以调试和维护，并且因指令操作复杂而增加机器周期，从而降低了系统性能。
- **1975年IBM公司开始研究指令系统的合理性问题，John Cocks提出精简指令系统计算机 RISC (Reduce Instruction Set Computer)。**

- 对CISC进行测试，发现一个事实：
  - 在程序中各种指令出现的频率悬殊很大，最常使用的是一些简单指令，这些指令占程序的80%，但只占指令系统的20%。而且在微程序控制的计算机中，占指令总数20%的复杂指令占用了控制存储器容量的80%。
- 1982年美国加州伯克利大学的RISC Ⅰ，斯坦福大学的MIPS，IBM公司的IBM801相继宣告完成，这些机器被称为第一代RISC机。

Top 10 80x86 Instructions

° Rank instruction Integer Average Percent total executed

1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

° Simple instructions dominate instruction frequency

（简单指令占主要部分，使用频率高！） back

2017年图灵奖得主

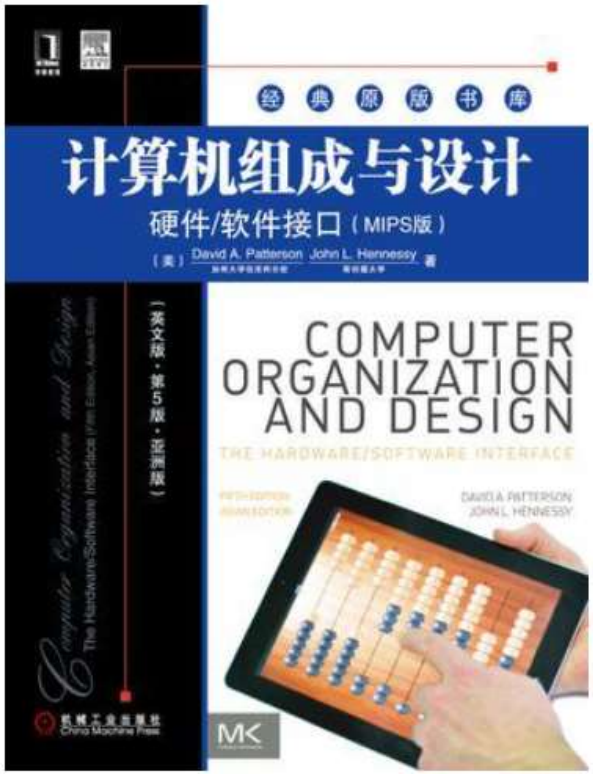
John Hennessy（左）和David Patterson 拿着他们合著的《计算机体系架构：量化研究方法》，照片的拍摄时间大约是1991年。

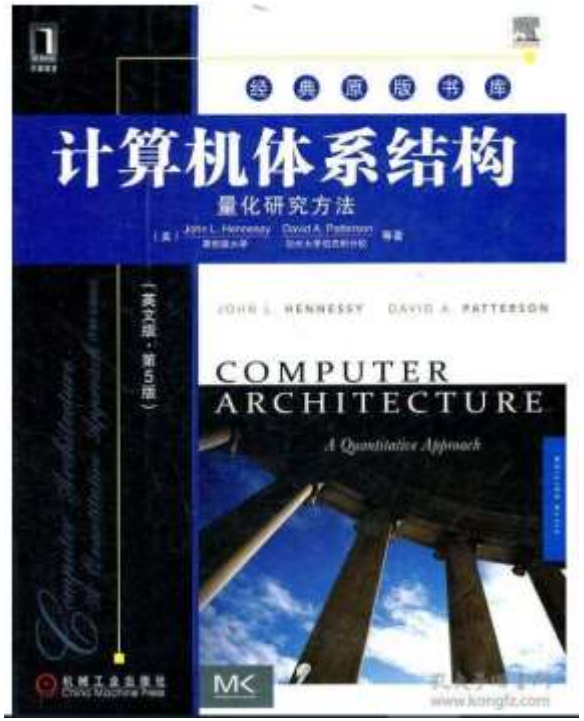
- The award praised them for "pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry"
- **开创了计算机体系结构设计和评估的系统的、量化的方法，对微处理器行业具有持久的影响力**

2017年图灵奖得主

- 他们提出的“RISC 指令集”
- 全球每年生产的160亿个微处理器中，99%都是RISC处理器，所有的智能手机、平板电脑、物联网设备中都有他们的技术。
- John Hennessy （1953年9月22日出生）
  - 1977年加入斯坦福任教
  - 1981年，组织研究人员致力于研究RISC（精简指令集计算机），并被称为“RISC之父”
  - 1984年，合作研制出MIPS计算机系统
  - 2000-2016年：任斯坦福大学校长
  - 目前，担任谷歌母公司Alphabet的首席执行官
- David Patterson（1947年11月16日出生）
  - 1976年加州大学伯克利分校计算机科学教授，退休后成为谷歌杰出工程师
  - Patterson以对RISC处理器设计的开创性贡献而闻名，创造了RISC这个术语，并领导了伯克利RISC项目
  - 他与Randy Katz一起开创了RAID存储的研究。

两位大师的著作——课本





Hennessy has a history of strong interest and involvement in college-level computer education. He co-authored, with David A. Patterson, two well-known books on computer architecture, *Computer Organization and Design: the Hardware/Software Interface* and *Computer Architecture: A Quantitative Approach*,which introduced the DLX RISC architecture. They have been widely used as textbooks for graduate and undergraduate courses since 1990

David Patterson对RISC的回顾

- 计算机发展之初，ROM比起RAM来说更便宜而且更快，所以并不存在片上缓存（cache）这个东西。在那个时候，复杂指令集（CISC）是主流的指令集架构。然而，随着RAM技术的发展，RAM速度越来越快，成本越来越低，因此在处理器上集成指令缓存成为可能。
- 同时，由于当时编译器的技术并不纯熟，程序都会直接以机器码或是汇编语言写成，为了减少程序设计师的设计时间，逐渐开发出单一指令，复杂操作的程序码，设计师只需写下简单的指令，再交由CPU去执行。
- 但是后来有人发现，整个指令集中，只有约20％的指令常常会被使用到，约占整个程序的80％；剩余80％的指令，只占整个程序的20％。
- 于是1979年，David Patterson教授提出了RISC的想法，主张硬件应该专心加速常用的指令，较为复杂的指令则利用常用的指令去组合。使用精简指令集（RISC）可以大大简化硬件的设计，从而使流水线设计变得简化，同时也让流水线可以运行更快。
- Patterson教授重申了评估处理器性能的指标，即程序运行时间。程序运行时间由几个因素决定，即程序指令数，平均指令执行周期数（CPI）以及时钟周期。程序指令数由程序代码，编译器以及ISA决定，CPI由ISA以及微架构决定，时钟周期由微架构以及半导体制造工艺决定。对于RISC，程序指令数较多，但是CPI远好于CISC，因此RISC比CISC更快。

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，
  其余指令的操作都在寄存器之间进行。
- CPU 中有多个 通用 寄存器
- 采用 流水技术，大部分指令在一个时钟周期内完成
- 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 7.5

- 系统指令 **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

四、RISC和CISC 的比较 7.5

类别 / 对比项目	CISC	RISC
指令系统	复杂，庞大	简单，精简
指令数目	一般大于 200 条	一般小于 100 条
指令字长	不固定	定长
可访存指令	不加限制	只有 Load/Store 指令
各种指令执行时间	相差较大	绝大多数在一个周期内完成
各种指令使用频度	相差很大	都比较常用
通用寄存器数量	较少	多
目标代码	难以用优化编译生成高效的目标代码程序	采用优化的编译程序，生成代码较为高效
控制方式	绝大多数为微程序控制	绝大多数为组合逻辑控制
指令流水线	可以通过一定方式实现	必须实现

单选题 1分

设置

[2009统考真题]下列关于RISC的说法中，错误的是(　　)。

A RISC 普遍采用微程序控制器

B RISC大多数指令在一个时钟周期内完成

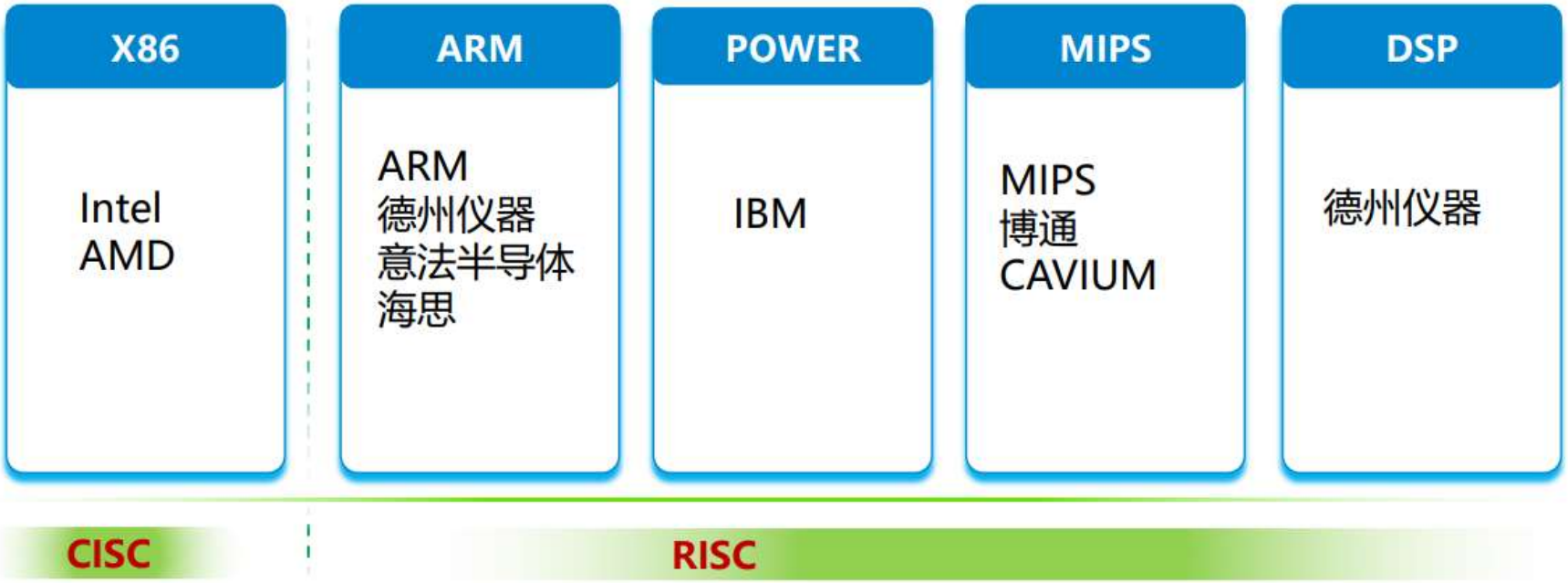
C RISC的内部通用寄存器数量相对CISC多

D RISC的指令数、寻址方式和指令格式种类相对CISC少

提交

处理器分类

架构类型	架构名称	推出公司	推出时间	主要授权商
CISC	X86	Intel, AMD	1978	海光，兆芯
RISC	ARM	ARM	1985	苹果，三星，英伟达，高通，海思，TI等
	MIPS	MIPS	1981	龙芯，炬力等
	POWER	IBM	1990	IBM



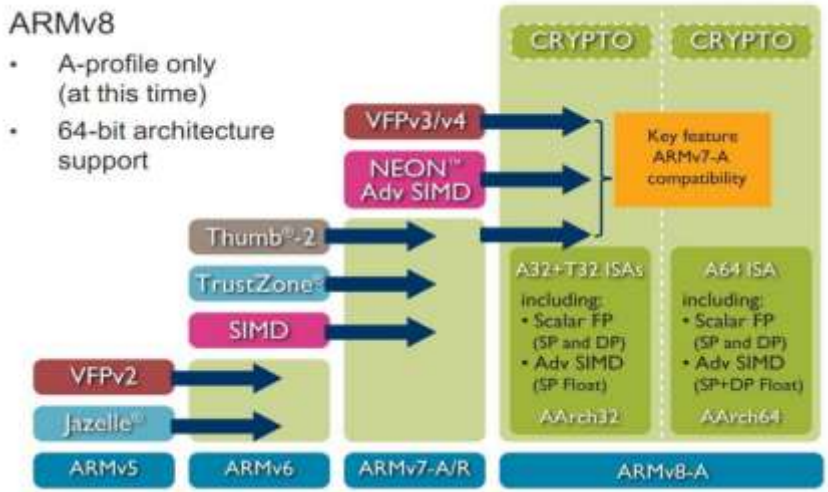
- CISC(Complex Instruction Set Computer)，复杂指令集。早期的CPU全部是CISC架构，它的设计目的是要用最少的机器语言指令来完成所需的计算任务。这种架构会增加CPU结构的复杂性和对CPU工艺的要求，但对于编译器的开发十分有利。
- RISC(Reduced Instruction Set Computer)，精简指令集。RISC架构要求软件来指定各个操作步骤。这种架构可以降低CPU的复杂性以及允许在同样的工艺水平下生产出功能更强大的CPU，但对于编译器的设计有更高的要求。

主流计算架构比较

	X86	ARM	POWER
指令集	• CISC, 复杂指令集	• RISC，精简指令集	• RISC，精简指令集
架构	• 重核架构，高性能高功耗	• 多核架构，均衡的性能功耗比	• 重核架构，高性能内核
生态	• 生态非常成熟，通用性强	• 生态正在快速发展与完备	• 聚焦大小型机和HPC
开放性	• 封闭架构，英特尔及AMD主导	• 开放平台，IP授权的商业模式	• 开放平台（2019.8），IBM主导

指令集：RISC vs CISC

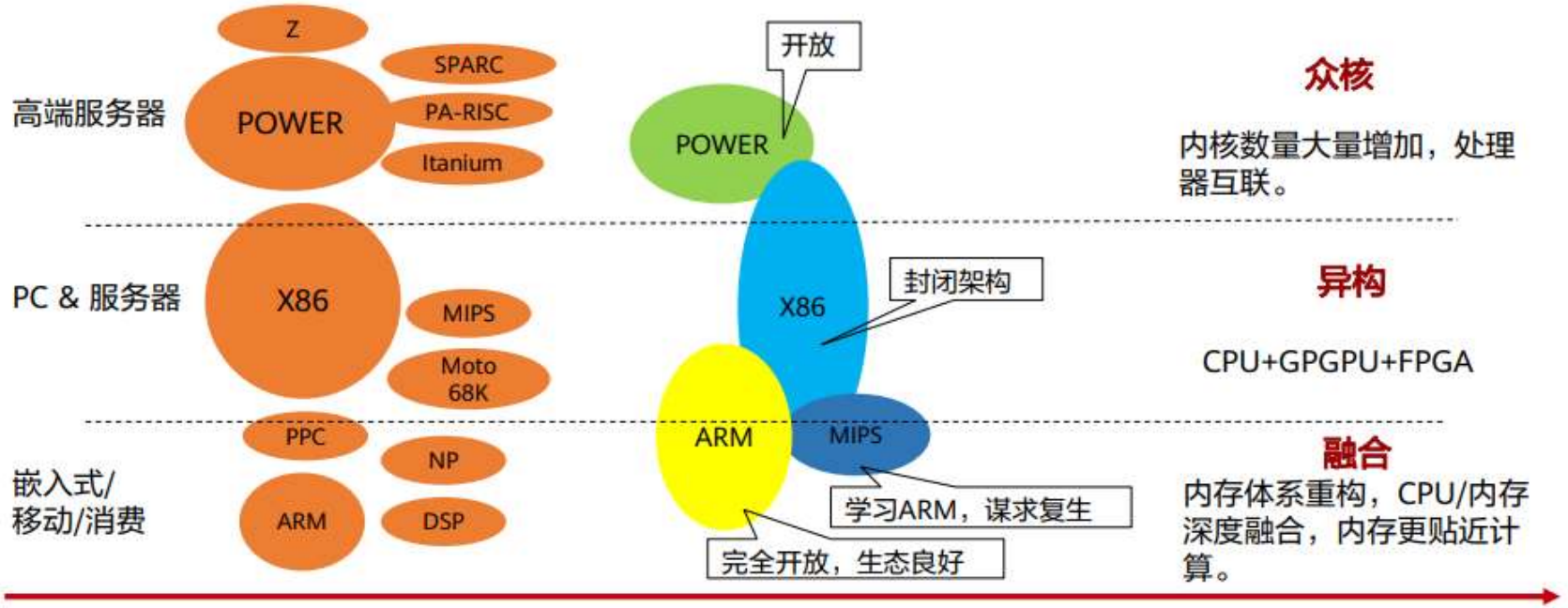
- ARM：使用精简指令集（RISC），大幅简化架构，仅保留所需要的指令，可以让整 个处理器更为简化，拥有小体积、高效能的特性；ARMv8架构支持64位操作，指令 32位，寄存器64位，寻址能力64位；指令集使用NEON扩展结构；
- X86：使用复杂指令集（CISC)，以增加处理器本身复杂度为代价，换取更高的性能； X86指令集从MMX，发展到了SSE，AVX；



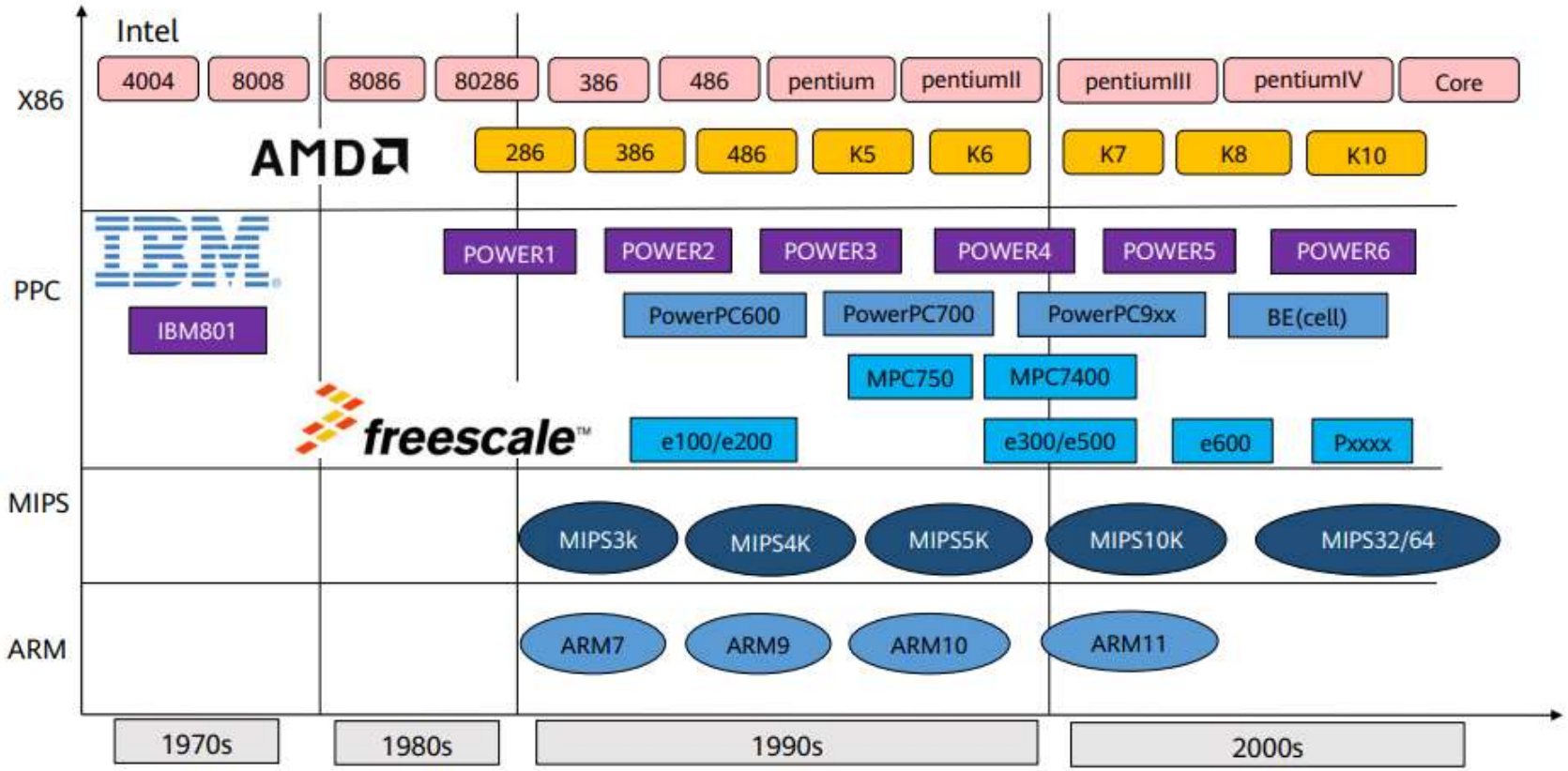
ARMv8架构的特点：

- 31个64位通用寄存器，原来架构只有15个通用寄存器；
- 新指令集支持64位运算，指令中的寄存器编码由4位扩充到5位；
- 新指令集仍然是32位，减少了条件执行指令，条件执行指令的4位编 码释放出来用于寄存器编码；
- 堆栈指针SP和程序指针PC都不再是通用寄存器了，同时推出了零值 寄存器（类似PowerPC的r0）；
- A64与A32的高级SIMD和FP相同；
- 高级SIMD与VFP共享浮点寄存器，支持128位宽的vector；
- 新增加解密指令。

处理器发展趋势

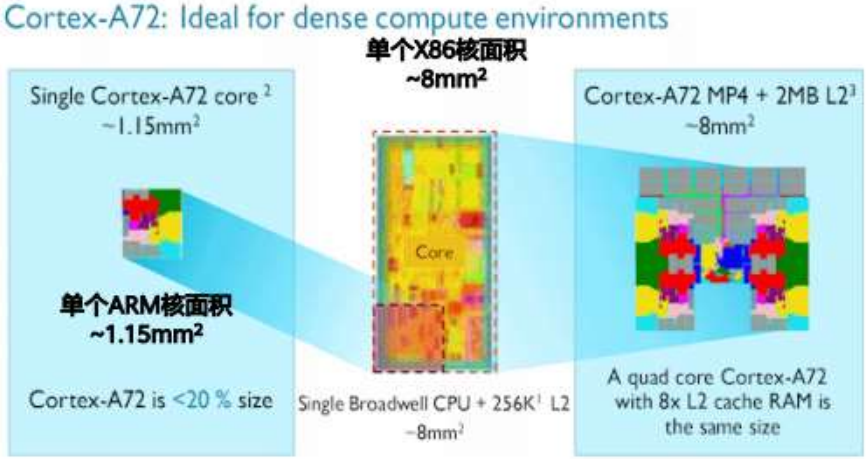

高端服务器
POWER
Z
SPARC
PA-RISC
Itanium
PC & 服务器
X86
MIPS
Moto 68K
嵌入式/
移动/消费
PPC
NP
ARM
DSP
开放
POWER
X86
封闭架构
ARM
MIPS
学习ARM，谋求复生
完全开放，生态良好
众核
内核数量大量增加，处理器互联。
异构
CPU+GPGPU+FPGA
融合
内存体系重构，CPU/内存深度融合，内存更贴近计算。
过去：架构众多，百花齐放
现在：生态成熟，架构垄断
未来：摩尔定律失效，寻求多方向突破

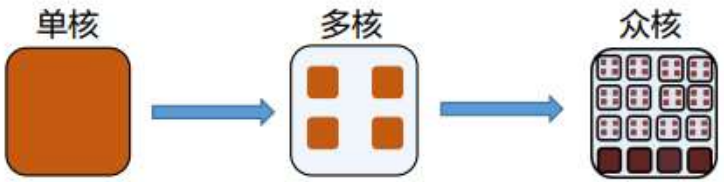
主流CPU发展路径


Intel
X86
4004
8008
8086
80286
386
486
pentium
pentiumII
pentiumIII
pentiumIV
Core
AMD
286
386
486
K5
K6
K7
K8
K10
IBM
PPC
POWER1
POWER2
POWER3
POWER4
POWER5
POWER6
IBM801
PowerPC600
PowerPC700
PowerPC9xx
BE(cell)
MPC750
MPC7400
freescale
e100/e200
e300/e500
e600
Pxxxx
MIPS
MIPS3k
MIPS4K
MIPS5K
MIPS10K
MIPS32/64
ARM
ARM7
ARM9
ARM10
ARM11
1970s
1980s
1990s
2000s

ARM提供更多计算核心

- 工艺、主频遇到瓶颈后，开始通过增加核数的方式来提升性能；
- 芯片的物理尺寸有限制，不能无限制的增加；
- ARM的众核横向扩展空间优势明显。





ARM处理器体系结构

- ARM内核工作模式:
  - 用户模式（user）：正常程序执行模式；
  - 快速中断模式（FIQ）：高优先级的中断产生会进入该种模式，用于高速通道传输；
  - 外部中断模式（IRQ）：低优先级中断产生会进入该模式，用于普通的中断处理；
  - 特权模式（Supervisor）：复位和软中断指令会进入该模式；
  - 数据访问中止模式（Abort）：当存储异常时会进入该模式；
  - 未定义指令中止模式（Undefined）：执行未定义指令会进入该模式；
  - 系统模式（System）：用于运行特权级操作系统任务；
  - 监控模式（Monitor）：可以在安全模式和非安全模式之间切换；

ARM处理器体系结构

- ARM体系结构的指令集（Instruction Set）
  - ARM指令集
  - Thumb指令集
  - Thumb-2指令集

ARM处理器体系结构

- ARM的微体系结构（Micro-architecture）
  - ARM处理器内核（Processor Core）
  - ARM处理器（Processor）
  - 基于ARM架构处理器的片上系统（SoC）

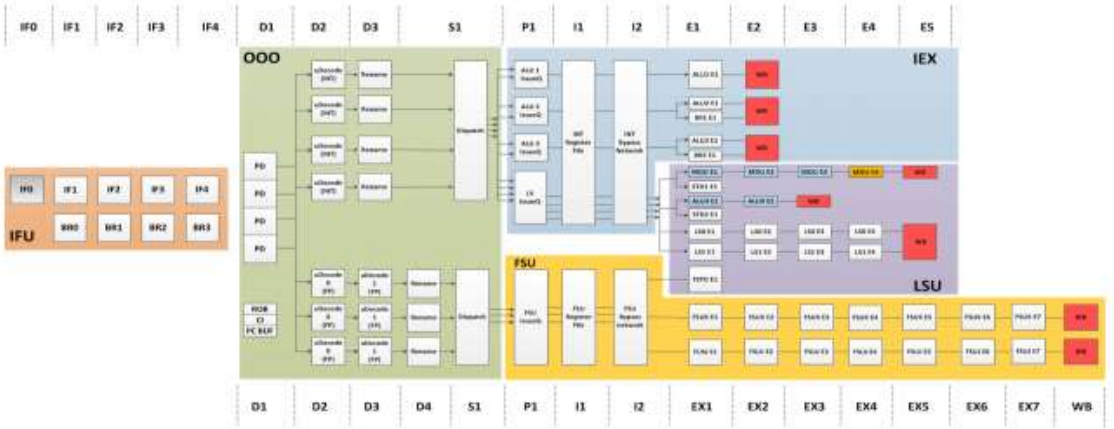
ARM公司授权体系

- ARM目前在全球拥有大约1000个授权合作商、320家伙伴，但是购买架构授权的厂家不超过20家，中国有华为、飞腾获得了架构授权。

ARM处理器体系结构

- ARM流水线的执行顺序：
  - 取指令（Fetch）：从存储器读取指令；
  - 译码（Decode）：译码以鉴别它是属于哪一条指令；
  - 执行（Execute）：将操作数进行组合以得到结果或存储器地址；
  - 缓冲/数据（Buffer/data）：如果需要，则访问存储器以存储数据；
  - 回写：（Write-back）：将结果写回到寄存器组中；

基于ARMv8的鲲鹏流水线技术



Taishan coreV110 Pipeline Architecture

- Branch预测和取指流水线解耦设计，取指流水线每拍最多可提供32Bytes指令供译码，分支预测流水线可以不受取指流水停顿影响，超前进行预测处理；
- 定浮点流水线分开设计，解除定浮点相互反压，每拍可为后端执行部件提供4条整型微指令及3条浮点微指令；
- 整型运算单元支持每拍4条ALU运算（含2条跳转）及1条乘除运算；
- 浮点及SIMD运算单元支持每拍2条ARM Neon 128bits 浮点及SIMD运算；
- 访存单元支持每拍2条读或写访存操作，读操作最快4拍完成，每拍访存带宽为2x128bits读及1x128bits写；

ARM处理器体系结构

- ARM处理器的分类：
  - ARM经典处理器（Classic Processors）；
  - ARM Cortex应用处理器；
    - 面向复杂操作系统和用户应用的Cortex-A（Applications，应用）系列
    - 针对实时处理和控制应用的Cortex-R（Real-time，实时）系列
    - 针对微控制器与低功耗应用优化的Cortex-M（Microcontroller）系列
  - ARM Cortex嵌入式处理器；
  - ARM专业处理器

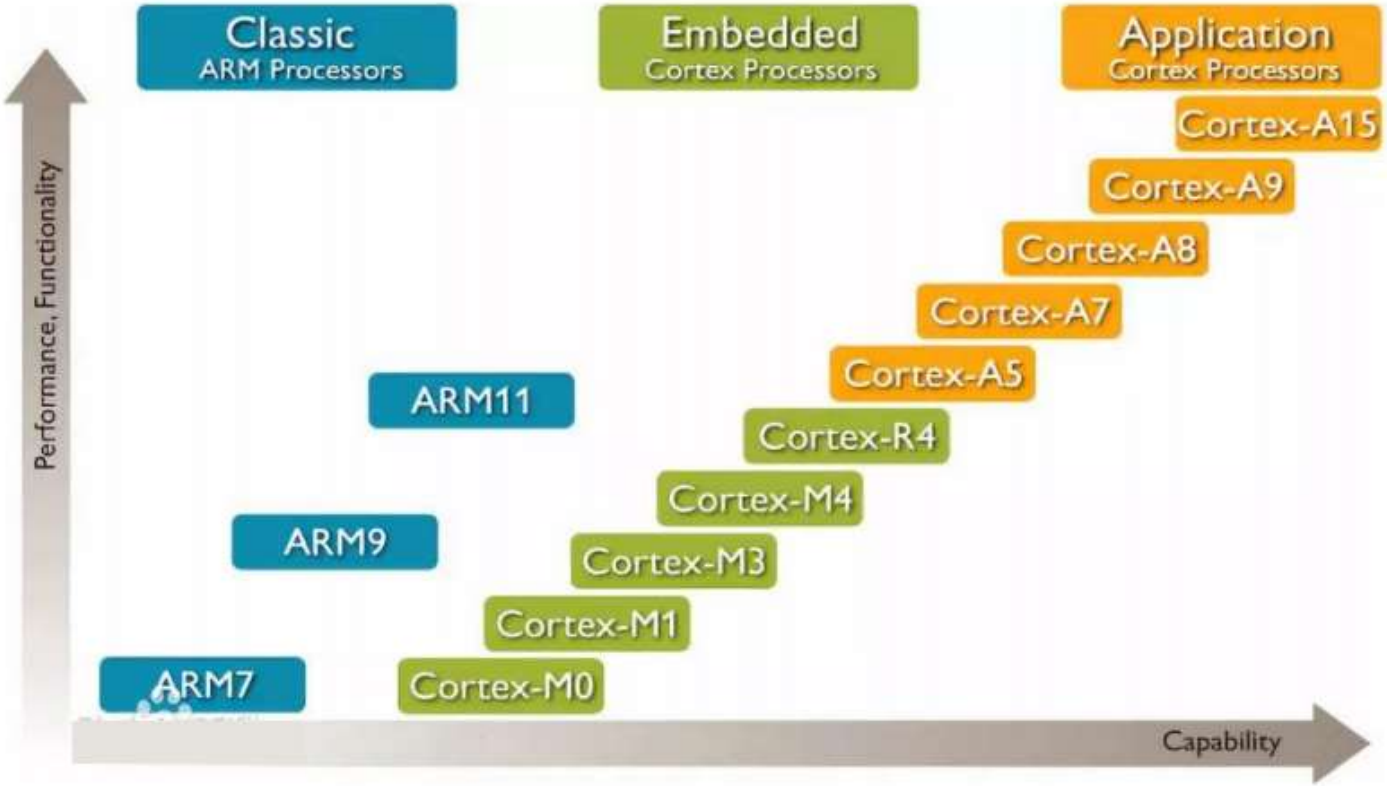
ARM 处理器系列命名规则

- 命名格式：ARM {x} {y} {z} {T} {D} {M} {I}
  - x：处理器系列，是共享相同硬件特性的一组处理器，如：ARM7TDMI、ARM740T 都属于 ARM7 系列
  - y：存储管理 / 保护单元
  - z：Cache
  - T：Thumb，Thumb16 位译码器
  - D：Debug，JTAG 调试器
  - M：Multipler，快速乘法器
  - I：Embedded ICE Logic，嵌入式跟踪宏单元

ARM架构发展史

架构	处理器家族
ARMv1	ARM1
ARMv2	ARM2、ARM3
ARMv3	ARM6、ARM7
ARMv4	StrongARM、ARM7TDMI、ARM9TDMI
ARMv5	ARM7EJ、ARM9E、ARM10E、XScale
ARMv6	ARM11、ARM Cortex-M
ARMv7	ARM Cortex-A、ARM Cortex-M、ARM Cortex-R
ARMv8	Cortex-A50[9]

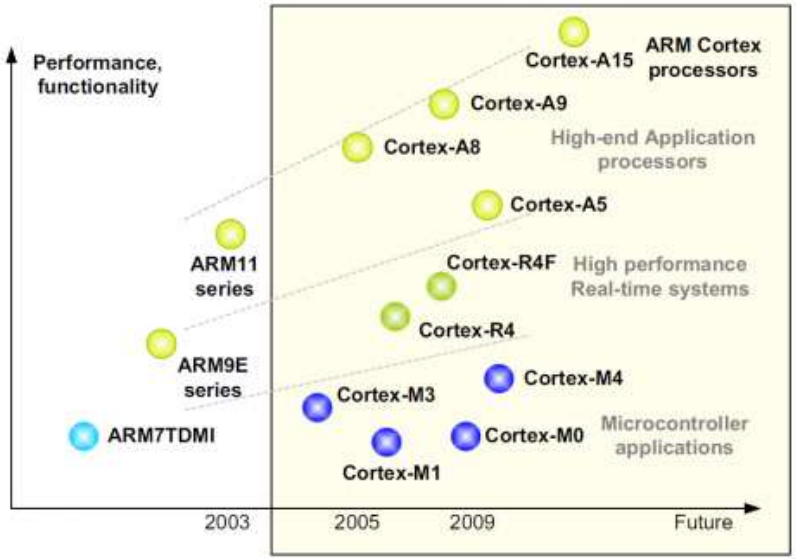
ARM架构发展史

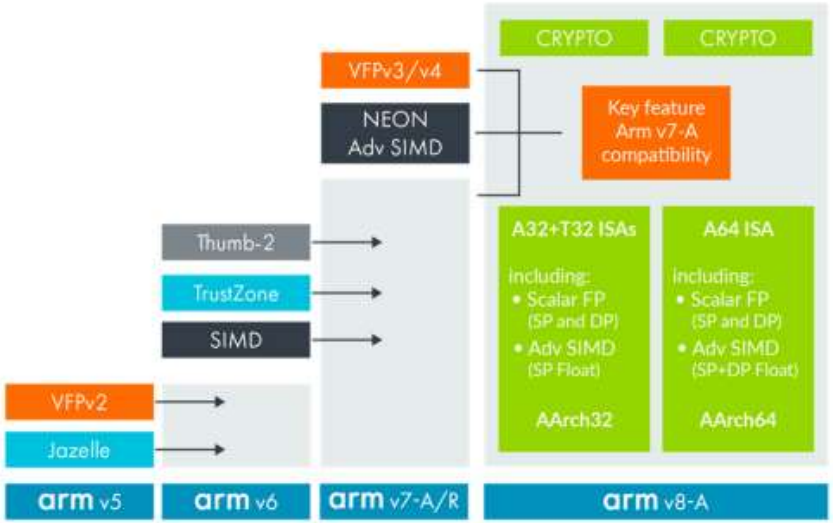


ARM架构发展史

- 从ARM v7开始，CPU命名为Cortex，并划分为A、R、M三大系列，分别为不同的市场提供服务；
  - A (Application)系列：应用型处理器，面向具有复杂软件操作系统的面向用户的应用，为手机、平板、AP等终端设备提供全方位的解决方案；
  - R (Real-Time)系列：实时高性能处理器，为要求可靠性、高可用性、容错功能、可维护性和实时响应的嵌入式系统提供高性能计算解决方案；
  - M (Microcontroller)系列：高能效、易于使用的处理器，主要用于通用低端，工业，消费电子领域微控制器。

ARM架构发展史


Performance, functionality
Cortex-A15
ARM Cortex processors
Cortex-A9
Cortex-A8
High-end Application processors
Cortex-A5
ARM11 series
Cortex-R4F
High performance Real-time systems
Cortex-R4
ARM9E series
Cortex-M4
Cortex-M3
ARM7TDMI
Cortex-M0
Microcontroller applications
Cortex-M1
2003
2005
2009
Future


CRYPTO
CRYPTO
VFPv3/v4
NEON Adv SIMD
Key feature Arm v7-A compatibility
Thumb-2
TrustZone
SIMD
A32+T32 ISAs
including:
• Scalar FP (SP and DP)
• Adv SIMD (SP Float)
AArch32
A64 ISA
including:
• Scalar FP (SP and DP)
• Adv SIMD (SP+DP Float)
AArch64
VFPv2
Jazelle
arm v5
arm v6
arm v7-A/R
arm v8-A

ARM服务器处理器的优势

- 低功耗一直以来都是ARM架构芯片最大的优势；
- ARM架构的芯片在成本、集成度方面也有较大的优势；
- 端、边、云全场景同构互联与协同；
- 更高的并发处理效率；
- 多元化的市场供应

华为鲲鹏处理器

1991 第一颗传输网络的ASIC芯片

2005 第一颗基于ARM的无线基站芯片

2009 第一颗基于ARM的移动端处理器

2014 Hi1612 第一颗基于ARM的64位处理器

2016 华为鲲鹏916 业界第一颗支持多路ARM 处理器

2019 华为鲲鹏920 业界第一颗7nm数据中心处理器

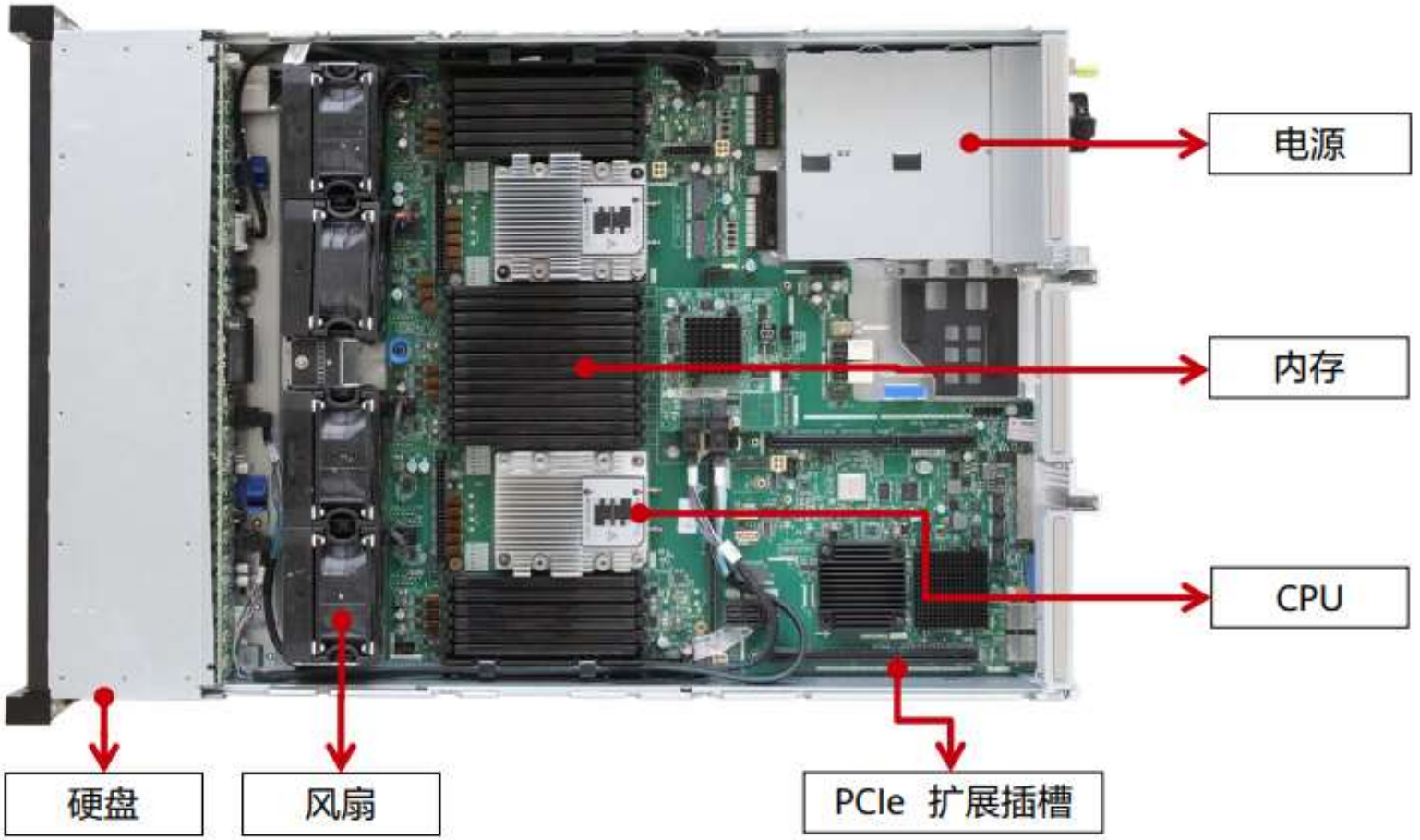
2021

华为鲲鹏930

2023

华为鲲鹏950

服务器内部视图



锲而舍之，朽木不折；
锲而不舍，金石可镂。

计 算 机 组 成 原 理

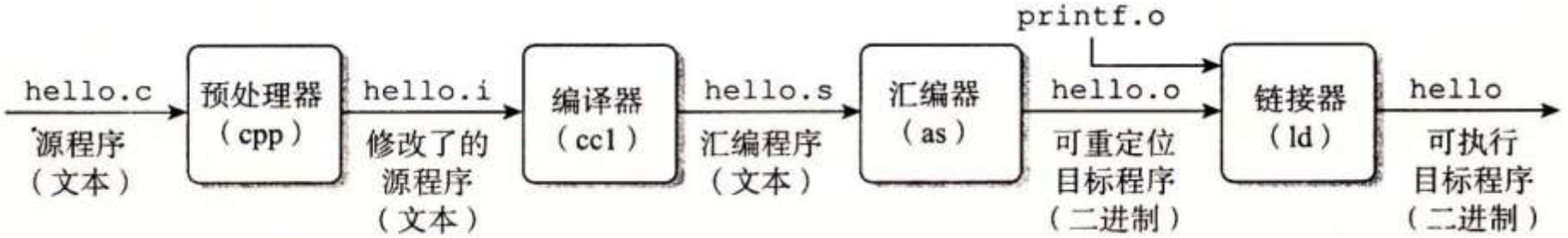
7.6 程序的机器级代码表示

7.6.1 编译器、汇编器和链路器基本概念

code/intro/hello.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6      return 0;
7  }
```

#	i	n	c	l	u	d	e	SP	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	SP	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	SP	SP	SP	SP	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	SP	w	o	r	l	d	\	n	"	)	;	\n	SP
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	SP	SP	r	e	t	u	r	n	SP	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	



7.6.1 编译器、汇编器和链路器基本概念

- 编译器:
- 编译器将高级语言转换成一种机器能理解的符号形式的**汇编语言程序**（assmbly language program）。
  - 编译器把高级语言翻译为机器语言
  - 得到*.s文件,这个是汇编语言程序
  - 不同的高级语言翻译的汇编语言相同

https://www.freesion.com/article/7404730325/
https://blog.csdn.net/qq_39918677/article/details/120372053

7.6.1 编译器、汇编器和链路器基本概念

- 汇编器:

汇编器将汇编语言程序转换成目标文件（object file），它包括机器语言指令、数据和指令正确放入内存所需要的信息。

  - 汇编器将*.s翻译成机器语言指令，把这些指令打包成可重定位目标程序。
  - 得到*.o文件,是一个二进制文件，它的字节码是机器语言指令，不再是字符。前面编译阶段都还有字符。

7.6.1 编译器、汇编器和链路器基本概念

- 链路器:

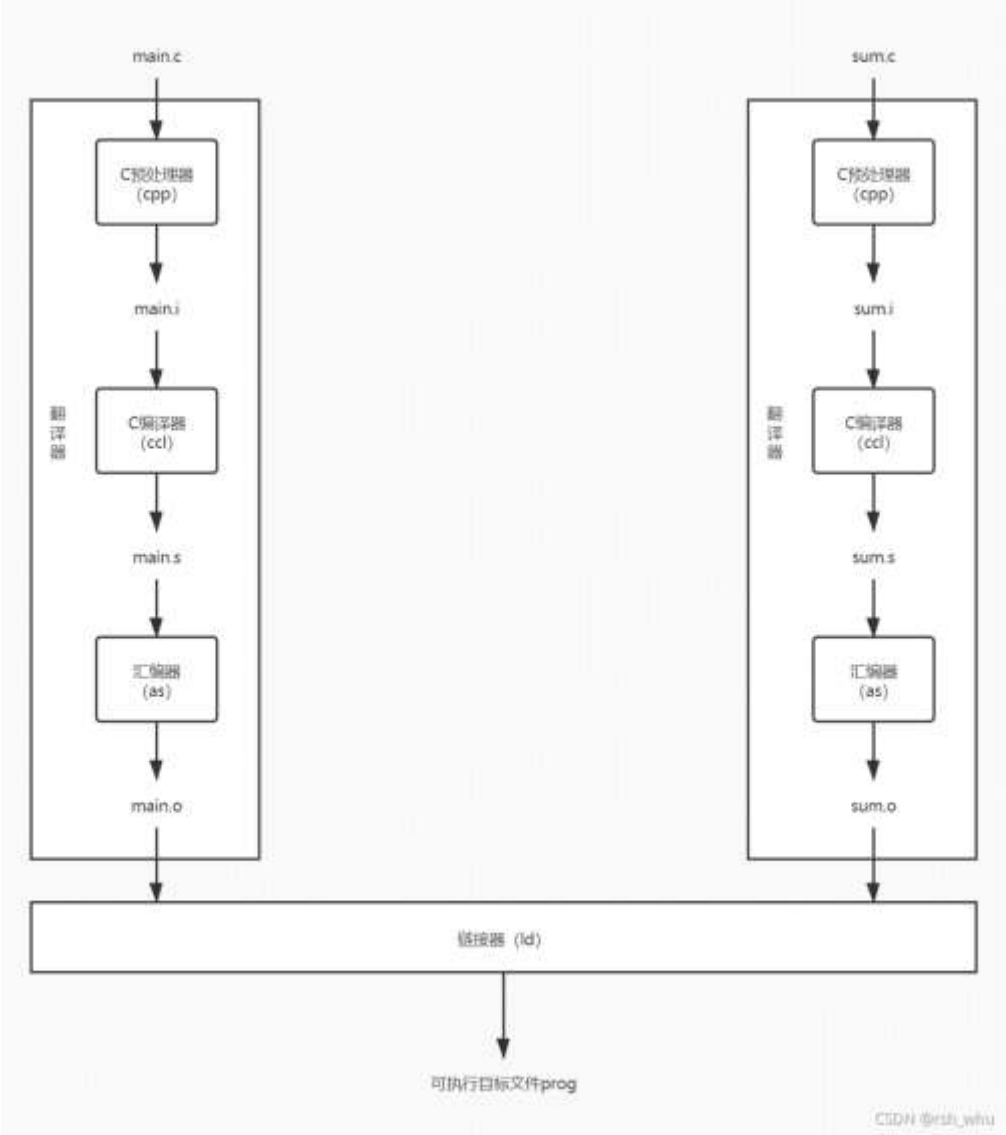
链接（lingking）是将各种代码和数据片段收集并组合成为一个大一文件的过程，这个文件可以加载（复制）到内存并执行。

可执行文件：一个具有目标文件格式的功能程序，不包含未解决的引用。它可以包含符号表和调试信息。

  - 链接器负责 .o 文件的合并。得到的是可执行目标文件。

- 链接器的工作分3个步骤
  - 将代码和数据模块象征性地放入内存。
  - 决定数据和指令标签的地址。
  - 修补内部和外部引用。

7.6.1 编译器、汇编器和链路器基本概念



7.6.2选择结构语句的机器级表示

- 汇编代码是机器代码的文本表示；
- 汇编代码格式：ATT（AT&T）格式和Intel格式；
- 两种格式混用；
- 汇编指令：数据传送、算术逻辑和控制流。

表 4.2　AT&T 格式指令和 Intel 格式指令的对比

AT&T 格式	Intel 格式	含 义
mov $100, %eax	mov eax, 100	100→R[eax]
mov %eax, %ebx	mov ebx, eax	R[eax]→R[ebx]
mov %eax, (%ebx)	mov [ebx], eax	R[eax]→M[R[ebx]]
mov %eax, -8(%ebp)	mov [ebp-8], eax	R[eax]→M[R[ebp]-8]]
lea 8(%edx,%eax,2), %eax	lea eax, [edx+eax*2+8]	R[edx]+R[eax]*2+8→R[eax]
movl %eax, %ebx	mov dword ptr ebx, eax	长度为 4 字节的 R[eax]→R[ebx]

注：R[r]表示寄存器 r 的内容，M[addr]表示主存单元 addr 的内容，→表示信息传送方向。

7.6.2选择结构语句的机器级表示

AT&T 格式和 Intel 格式。它们的区别主要体现如下：

① AT&T 格式的指令只能用小写字母，而 Intel 格式的指令对大小写不敏感。

② 在 AT&T 格式中，第一个为源操作数，第二个为目的操作数，方向从左到右，合乎自然；在 Intel 格式中，第一个为目的操作数，第二个为源操作数，方向从右向左。

③ 在 AT&T 格式中，寄存器需要加前缀“%”，立即数需要加前缀“$”；在 Intel 格式中，寄存器和立即数都不需要加前缀。

④ 在内存寻址方面，AT&T 格式使用“(”和“)”，而 Intel 格式使用“[”和“]”。

⑤ 在处理复杂寻址方式时，例如 AT&T 格式的内存操作数“disp(base, index, scale)”分别表示偏移量、基址寄存器、变址寄存器和比例因子，如“8(%edx, %eax, 2)”表示操作数为 M[R[edx]+R[eax]*2+8]，其对应的 Intel 格式的操作数为“[edx+eax*2+8]”。

⑥ 在指定数据长度方面，AT&T 格式指令操作码的后面紧跟一个字符，表明操作数大小，“b”表示 byte（字节）、“w”表示 word（字）或“l”表示 long（双字）。Intel 格式也有类似的语法，它在操作码后面显式地注明 byte ptr、word ptr 或 dword ptr。

注意：由于 32 或 64 位体系结构都是由 16 位扩展而来的，因此用 word（字）表示 16 位。

7.6.2选择结构语句的机器级表示

- 常用的选择语句有 if-then、if-then-else、case（或switch）等
- 编译器通过条件码（标志位）设置指令和各类转移指令来实现程序中的选择结构语句。

（1）条件码（标志位）：描述最近的算术或逻辑运算操作的属性。

- CF：进（借）位标志；CF=1，有进/借位
- ZF：零标志；ZF=1，结果为零
- SF：符号标志；SF=1，为负
- OF：溢出标志；OF=1，溢出

常见的算术逻辑运算会设置（add,sub,imul等）

cmp、test：只设置条件码，不更新目的寄存器。

7.6.2选择结构语句的机器级表示

- 常用的选择语句有 if-then、if-then-else、case（或switch）等

（1）if 语句

```
long lt_cnt = 0;
long ge_cnt = 0;

long absdiff_se(long x, long y)
{
    long result;
    if (x < y) {
        lt_cnt++;
        result = y - x;
    }
    else {
        ge_cnt++;
        result = x - y;
    }
    return result;
}
```

a）原始的C语言代码

```
1   long gotodiff_se(long x, long y)
2   {
3       long result;
4       if (x >= y)
5           goto x_ge_y;
6       lt_cnt++;
7       result =  y - x;
8       return result;
9    x_ge_y:
10      ge_cnt++;
11      result = x - y;
12      return result;
13  }
```

b）与之等价的goto版本

7.6.2选择结构语句的机器级表示

```
long lt_cnt = 0;
long ge_cnt = 0;

long absdiff_se(long x, long y)
{
    long result;
    if (x < y) {
        lt_cnt++;
        result = y - x;
    }
    else {
        ge_cnt++;
        result = x - y;
    }
    return result;
}
```

```
1  long gotodiff_se(long x, long y)
2  {
3      long result;
4      if (x >= y)
5          goto x_ge_y;
6      lt_cnt++;
7      result = y - x;
8      return result;
9   x_ge_y:
10     ge_cnt++;
11     result = x - y;
12     return result;
13 }
```

```
    long absdiff_se(long x, long y)
    x in %rdi, y in %rsi
1   absdiff_se:
2     cmpq    %rsi, %rdi          Compare x:y
3     jge     .L2                 If >= goto x_ge_y
4     addq    $1, lt_cnt(%rip)    lt_cnt++
5     movq    %rsi, %rax
6     subq    %rdi, %rax          result = y - x
7     ret                         Return
8   .L2:                        x_ge_y:
9     addq    $1, ge_cnt(%rip)    ge_cnt++
10    movq    %rdi, %rax
11    subq    %rsi, %rax          result = x - y
12    ret                         Return
```

c）产生的汇编代码

7.6.2选择结构语句的机器级表示

C语言中的 if-else 语句的通用形式模板如下：

```
if (test-expr)
    then-statement
else
    else-statement
```

这里 *test-expr* 是一个整数表达式，它的取值为 0(解释为“假”)或者为非 0(解释为“真”)。两个分支语句中(*then-statement* 或 *else-statement*)只会执行一个。

对于这种通用形式，汇编实现通常会使用下面这种形式，这里，我们用 C 语法来描述控制流：

```
    t = test-expr;
    if (!t)
        goto false;
    then-statement
    goto done;
false:
    else-statement
done:
```

也就是，汇编器为 *then-statement* 和 *else-statement* 产生各自的代码块。它会插入条件和无条件分支，以保证能执行正确的代码块。

7.6.2选择结构语句的机器级表示

（2）switch 语句

```
void switch_eg(long x, long n,
               long *dest)
{
    long val = x;

    switch (n) {

    case 100:
        val *= 13;
        break;

    case 102:
        val += 10;
        /* Fall through */

    case 103:
        val += 11;
        break;

    case 104:
    case 106:
        val *= val;
        break;

    default:
        val = 0;
    }
    *dest = val;
}
```

a）switch语句

```
1  void switch_eg_impl(long x, long n,
2                      long *dest)
3  {
4      /* Table of code pointers */
5      static void *jt[7] = {
6          &&loc_A, &&loc_def, &&loc_B,
7          &&loc_C, &&loc_D, &&loc_def,
8          &&loc_D
9      };
10     unsigned long index = n - 100;
11     long val;
12
13     if (index > 6)
14         goto loc_def;
15     /* Multiway branch */
16     goto *jt[index];
17
18  loc_A:    /* Case 100 */
19     val = x * 13;
20     goto done;
21  loc_B:    /* Case 102 */
22     x = x + 10;
23     /* Fall through */
24  loc_C:    /* Case 103 */
25     val = x + 11;
26     goto done;
27  loc_D:    /* Cases 104, 106 */
28     val = x * x;
29     goto done;
30  loc_def:  /* Default case */
31     val = 0;
32  done:
33     *dest = val;
34 }
```

b）翻译到扩展的C语言

```
    void switch_eg(long x, long n, long *dest)
    x in %rdi, n in %rsi, dest in %rdx
1   switch_eg:
2     subq    $100, %rsi          Compute index = n-100
3     cmpq    $6, %rsi            Compare index:6
4     ja      .L8                 If >, goto loc_def
5     jmp     *.L4(,%rsi,8)       Goto *jt[index]
6   .L3:                          loc_A:
7     leaq    (%rdi,%rdi,2), %rax   3*x
8     leaq    (%rdi,%rax,4), %rdi   val = 13*x
9     jmp     .L2                 Goto done
10  .L5:                          loc_B:
11    addq    $10, %rdi             x = x + 10
12  .L6:                          loc_C:
13    addq    $11, %rdi             val = x + 11
14    jmp     .L2                 Goto done
15  .L7:                          loc_D:
16    imulq   %rdi, %rdi            val = x * x
17    jmp     .L2                 Goto done
18  .L8:                          loc_def:
19    movl    $0, %edi              val = 0
20  .L2:                          done:
21    movq    %rdi, (%rdx)          *dest = val
22    ret                         Return
```

图 3-23　图 3-22 中 switch 语句示例的汇编代码

7.6.3 循环结构语句的机器级表示

(1)do while循环

do-while 语句的通用形式如下：

```
do
    body-statement
    while (test-expr);
```

这种通用形式可以被翻译成如下所示的条件和 goto 语句：

```
loop:
    body-statement
    t = test-expr;
    if (t)
       goto loop;
```

7.6.3 循环结构语句的机器级表示

```
long fact_do(long n)
{
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

a）C代码

```
long fact_do_goto(long n)
{
    long result = 1;
 loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

b）等价的goto版本

```
    long fact_do(long n)
    n in %rdi
1   fact_do:
2     movl    $1, %eax        Set result = 1
3   .L2:                    loop:
4     imulq   %rdi, %rax      Compute result *= n
5     subq    $1, %rdi        Decrement n
6     cmpq    $1, %rdi        Compare n:1
7     jg      .L2             If >, goto loop
8     rep; ret                Return
```

c）对应的汇编代码

7.6.3 循环结构语句的机器级表示

(2)while循环

while 语句的通用形式如下：

```
while (test-expr)
    body-statement
```

通用的 while 循环格式翻译到 goto 代码：

```
    goto test;
loop:
    body-statement
test:
    t = test-expr;
    if (t)
        goto loop;
```

7.6.3 循环结构语句的机器级表示

(2)while循环

while 语句的通用形式如下：

```
while(test_expr)
    body_statement
```

与 do-while 的不同之处在于，第一次执行 body_statement 之前，就会测试 test_expr 的值，循环有可能中止。GCC 通常会将其翻译成条件分支加 do-while 循环的方式。

用如下模板来表达这种方法，将通用的 while 循环格式翻译成 do-while 循环：

```
    t=test_expr;
    if(!t)
        goto done;
    do
        body_statement
        while(test_expr);
done:
```

7.6.3 循环结构语句的机器级表示

(2)while循环

相应地，进一步将它翻译成 goto 语句：

```
    t=test_expr;
    if(!t)
        goto done;
loop:
    body_statement
    t=test_expr;
    if(t)
        goto loop;
done:
```

7.6.3 循环结构语句的机器级表示

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

a）C代码

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
 loop:
    result *= n;
    n = n-1;
 test:
    if (n > 1)
        goto loop;
    return result;
}
```

b）等价的goto版本

```
  long fact_while(long n)
  n in %rdi
fact_while:
  movl    $1, %eax        Set result = 1
  jmp     .L5             Goto test
.L6:                    loop:
  imulq   %rdi, %rax      Compute result *= n
  subq    $1, %rdi        Decrement n
.L5:                    test:
  cmpq    $1, %rdi        Compare n:1
  jg      .L6             If >, goto loop
  rep; ret                Return
```

c）对应的汇编代码

7.6.3 循环结构语句的机器级表示

(3) for循环

for 循环的通用形式如下：

```
for (init-expr; test-expr; update-expr)
    body-statement
```

C 语言标准说明(有一个例外，练习题 3.29 中有特别说明)，这样一个循环的行为与下面这段使用 while 循环的代码的行为一样：

```
init-expr;
while (test-expr) {
    body-statement
    update-expr;
}
```

7.6.3 循环结构语句的机器级表示

(3) for循环

for 循环的通用形式如下：

```
for(init_expr; test_expr; update_expr)
    body_statement
```

这个 for 循环的行为与下面这段 while 循环代码的行为一样：

```
init_expr;
while(test_expr){
    body_statement
    update_expr;
}
```

进一步把它翻译成 goto 语句：

```
    init_expr;
    t=test_expr;
    if(!t)
        goto done;
loop:
    body_statement
    update_expr;
    t=test_expr;
    if(t)
        goto loop;
done:
```

7.6.3 循环结构语句的机器级表示

GCC 为 for 循环产生的代码是 while 循环的两种翻译之一，这取决于优化的等级。也就是，跳转到中间策略会得到如下 goto 代码：

```
    init-expr;
    goto test;

loop:
    body-statement
    update-expr;
test:
    t = test-expr;
    if (t)
        goto loop;
```

而 guarded-do 策略得到：

```
    init-expr;
    t = test-expr;
    if (!t)
        goto done;
loop:
    body-statement
    update-expr;
    t = test-expr;
    if (t)
        goto loop;
done:
```

作为一个示例，考虑用 for 循环写的阶乘函数：

```
long fact_for(long n)
{
    long i;
    long result = 1;
    for (i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

7.6.3 循环结构语句的机器级表示

下面是一个用 for 循环写的自然数求和的函数：

```
int nsum_for(int n){
    int i;
    int result = 0;
    for(i=1;i<=n;i++)
        result +=i;
    return result;
}
```

这段代码中的 for 循环的不同组成部分如下：

```
init_expr          i=1
test_expr          i<=n
update_expr        i++
body_statement     result +=i
```

通过替换前面给出的模板中的相应位置，很容易将 for 循环转换为 while 或 do-while 循环。将这个阶乘函数翻译为 goto 语句代码后，不难得出其过程体的汇编代码：

```
    movl    8(%ebp),%ecx        #R[ecx]←M[R[ebp]+8]，即 R[ecx]=n
    movl    $0,%eax             #R[eax]←0，即 result=0
    movl    $1,%edx             #R[edx]←1，即 i=1
    cmp     %ecx,%edx           #Compare R[ecx]:R[edx]，即比较 i:n
    jg  .L2                     #If greater，转跳到 L2 执行
    .L1:                        #loop:
    addl    %edx,%eax           #R[eax]←R[eax]+R[edx]，即 result +=i
    addl    $1,%edx             #R[edx]←R[edx]+1，即 i++
    cmpl    %ecx,%edx           #比较%ecx 和%edx，即比较 i:n
    jle .L1                     #If less or equal，转跳到 L1 执行
    .L2:
```

已知 n 对应的实参已被压入调用函数的栈帧，其对应的存储地址为 R[ebp]+8，过程 nsum_for 中的局部变量 i 和 result 被分别分配到寄存器 EDX 和 EAX 中，返回参数在 EAX 中。

7.6.4 过程/函数调用对应的机器级表示

- 常用的选择语句有 if-then、if-then-else、case（或switch）等
- 编译器通过条件码（标志位）设置指令和各类转移指令来实现程序中的选择结构语句。

（1）条件码（标志位）：描述最近的算术或逻辑运算操作的属性。

- CF：进（借）位标志；CF=1，有进/借位
- ZF：零标志；ZF=1，结果为零
- SF：符号标志；SF=1，为负
- OF：溢出标志；OF=1，溢出

常见的算术逻辑运算会设置（add,sub,imul等）

cmp、test：只设置条件码，不更新目的寄存器。