

计算机组成原理

# 第7章 指令系统

系统结构研究所



## 四. 指令系统

### (一) 指令格式

- 1、指令的基本格式
- 2、定长操作码指令格式
- 3、扩展操作码指令格式

### (二) 指令的寻址方式

- 1、有效地址的概念
- 2、数据寻址和指令寻址
- 3、常见寻址方式

### (三) CISC和RISC的基本概念

# Contents

- 1 7.1 机器指令
- 2 7.2 操作数类型和操作类型
- 3 7.3 寻址方式
- 4 7.4 指令格式举例
- 5 7.5 RISC 技术

# 7.1 机器指令

## 一、指令格式

**指令：**就是让计算机执行某种操作的命令。

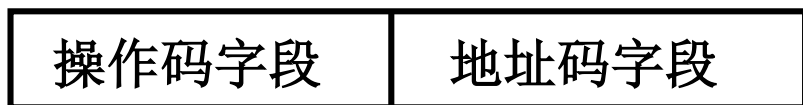
**指令系统：**一台计算机中所有机器指令的集合称为这台计算机的指令系统。

**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。

# 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。

1. 操作码 指令应该执行什么性质的操作和具有何种功能。

(1) 长度固定

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

7.1

操作码的位数随地址数的减少而增加

	OP	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	
4 位操作码	0000 0001 ⋮ 1110	A <sub>1</sub> A <sub>1</sub> ⋮ A <sub>1</sub>	A <sub>2</sub> A <sub>2</sub> ⋮ A <sub>2</sub>	A <sub>3</sub> A <sub>3</sub> ⋮ A <sub>3</sub>	最多15条三地址指令
8 位操作码	1111 1111 ⋮ 1111	0000 0001 ⋮ 1110	A <sub>2</sub> A <sub>2</sub> ⋮ A <sub>2</sub>	A <sub>3</sub> A <sub>3</sub> ⋮ A <sub>3</sub>	最多15条二地址指令
12 位操作码	1111 1111 ⋮ 1111	1111 1111 ⋮ 1111	0000 0001 ⋮ 1110	A <sub>3</sub> A <sub>3</sub> ⋮ A <sub>3</sub>	最多15条一地址指令
16 位操作码	1111 1111 ⋮ 1111	1111 1111 ⋮ 1111	1111 1111 ⋮ 1111	0000 0001 ⋮ 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

	OP	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>
4 位操作码	0000	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>
	0001	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>
	⋮	⋮	⋮	⋮
	1110	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>
8 位操作码	1111	0000	A <sub>2</sub>	A <sub>3</sub>
	1111	0001	A <sub>2</sub>	A <sub>3</sub>
	⋮	⋮	⋮	⋮
	1111	1110	A <sub>2</sub>	A <sub>3</sub>
12 位操作码	1111	1111	0000	A <sub>3</sub>
	1111	1111	0001	A <sub>3</sub>
	⋮	⋮	⋮	⋮
	1111	1111	1110	A <sub>3</sub>
16 位操作码	1111	1111	1111	0000
	1111	1111	1111	0001
	⋮	⋮	⋮	⋮
	1111	1111	1111	1111

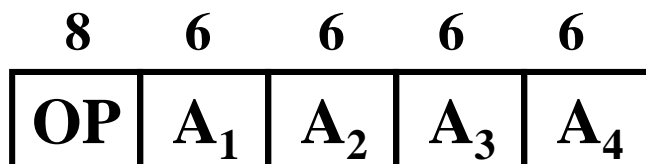
三地址指令操作码  
每减少一种可多构成  
2<sup>4</sup> 种二地址指令

二地址指令操作码  
每减少一种可多构成  
2<sup>4</sup> 种一地址指令

## 2. 地址码

7.1

### (1) 四地址



A<sub>1</sub> 第一操作数地址

A<sub>2</sub> 第二操作数地址

A<sub>3</sub> 结果的地址

A<sub>4</sub> 下一条指令地址

$(A_1) \text{ OP } (A_2) \longrightarrow A_3$

设指令字长为 32 位

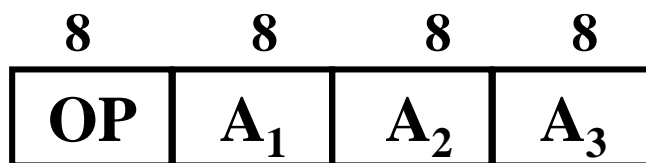
操作码固定为 8 位

4 次访存

寻址范围  $2^6 = 64$

若 PC 代替 A<sub>4</sub>

### (2) 三地址



$(A_1) \text{ OP } (A_2) \longrightarrow A_3$

4 次访存

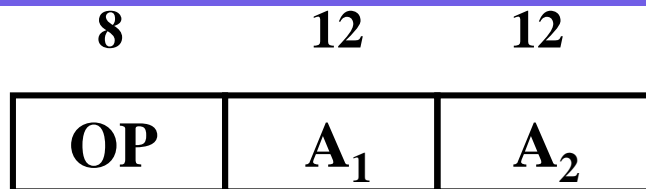
寻址范围  $2^8 = 256$

若 A<sub>3</sub> 用 A<sub>1</sub> 或 A<sub>2</sub> 代替



### (3) 二地址

7.1



或  
(A<sub>1</sub>) OP (A<sub>2</sub>)  $\longrightarrow$  A<sub>1</sub>  
(A<sub>1</sub>) OP (A<sub>2</sub>)  $\longrightarrow$  A<sub>2</sub>

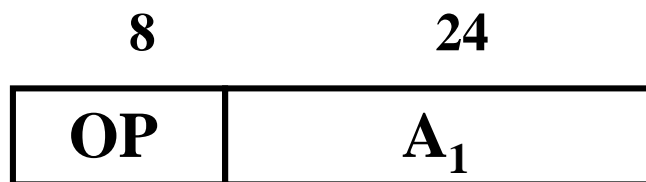
4 次访存

寻址范围  $2^{12} = 4 \text{ K}$

若结果存于 ACC 3次访存

若ACC 代替 A<sub>1</sub> (或A<sub>2</sub>)

### (4) 一地址



2 次访存

(ACC) OP (A<sub>1</sub>)  $\longrightarrow$  ACC

寻址范围  $2^{24} = 16 \text{ M}$

### (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。

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

### 2. 指令字长 可变

按字节的倍数变化

### ➤ 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

## 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

大端小端方式

存储器中的数据存放：边界对齐

### 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**

### 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

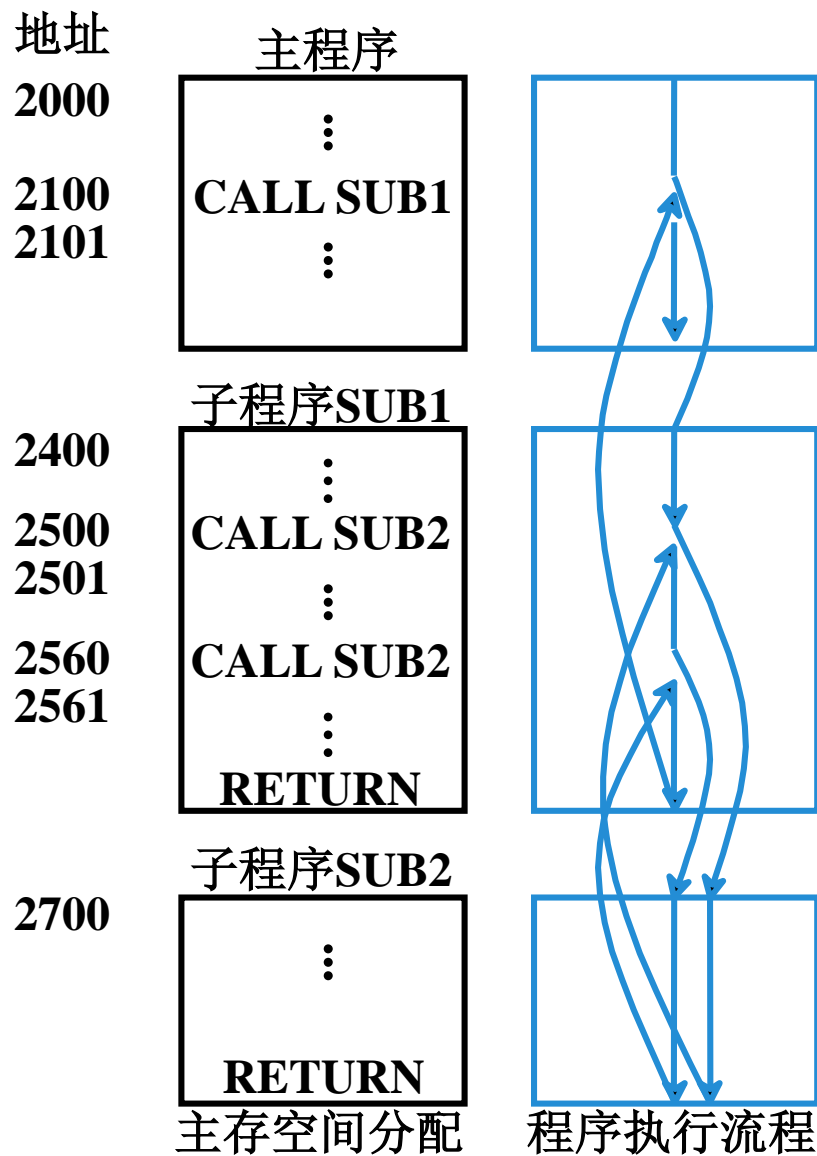
300  
⋮  
305  
306  
→ 307

完成触发器

**SKP DZ D = 0 则跳**

### (3) 调用和返回

7.2



## (4) 陷阱 (Trap) 与陷阱指令

7.2

### 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

8位常数，表示中断类型

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       $\longrightarrow$       CPU 的寄存器

如      **IN AX, m**      **IN AX, DX**

出      CPU 的寄存器       $\longrightarrow$       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

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

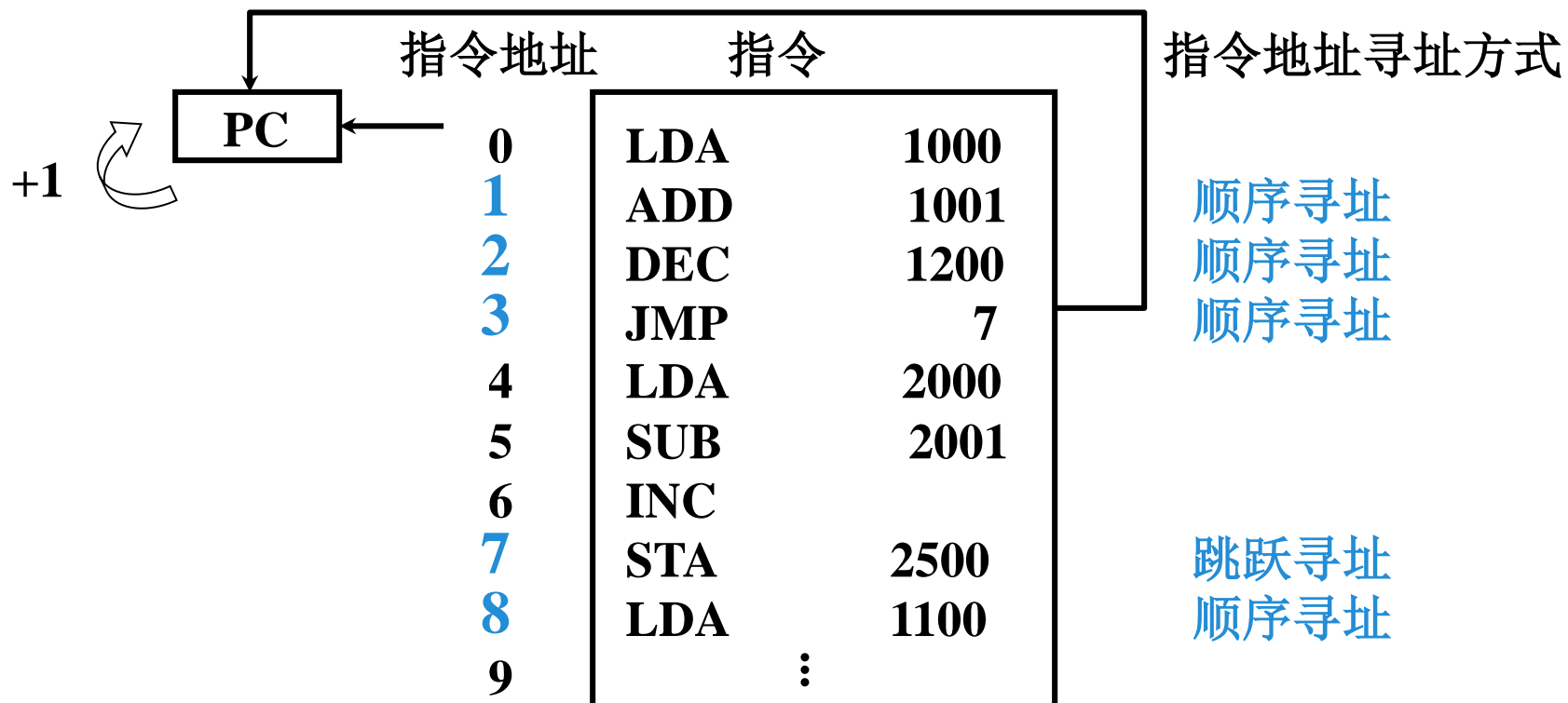
- (1) 直接寻址 2000
- (2) 间接寻址 1000
- (3) 寄存器间接寻址 R
- (4) 相对寻址 -1000

## 7.3 寻址方式

### 一、指令寻址

顺序  $(PC) + 1 \longrightarrow PC$

跳跃 由转移指令指出



操作码	寻址特征	形式地址 A
-----	------	--------

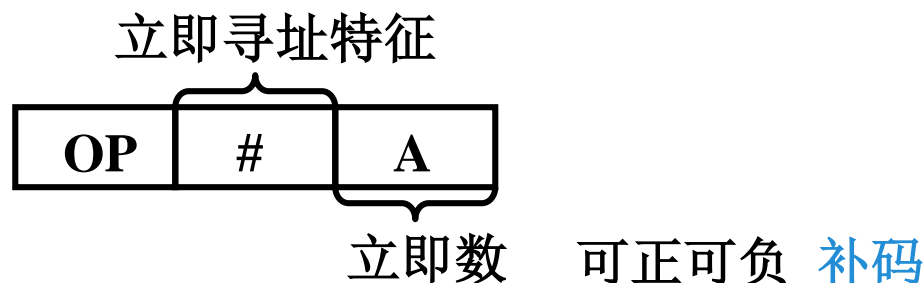
形式地址          指令字中的地址

有效地址          操作数的真实地址

约定    指令字长 = 存储字长 = 机器字长

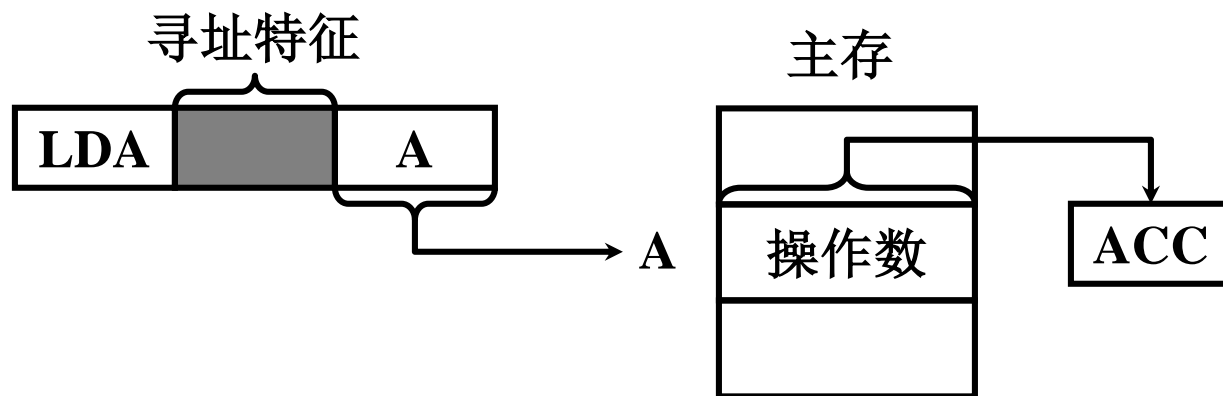
### 1. 立即寻址

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

$EA = A$       有效地址由形式地址直接给出

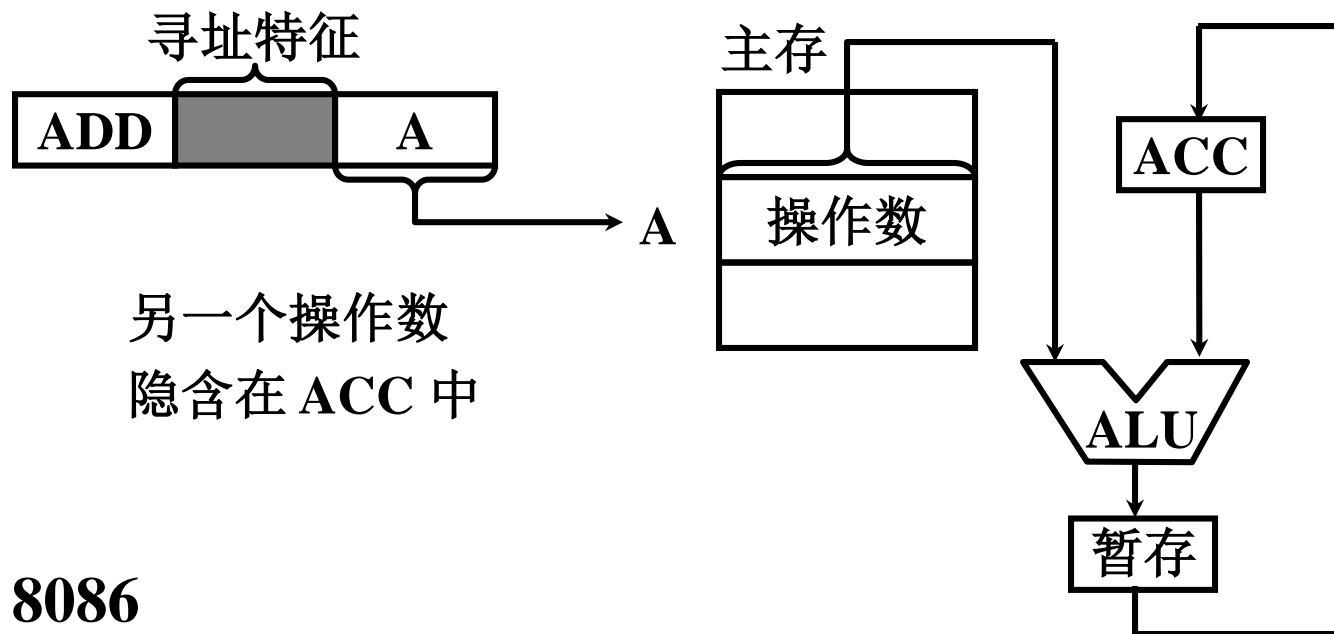


- 执行阶段访问一次存储器
- A 的位数决定了该指令操作数的寻址范围
- 操作数的地址不易修改（必须修改A）

### 3. 隐含寻址

7.3

#### 操作数地址隐含在操作码中



如 8086

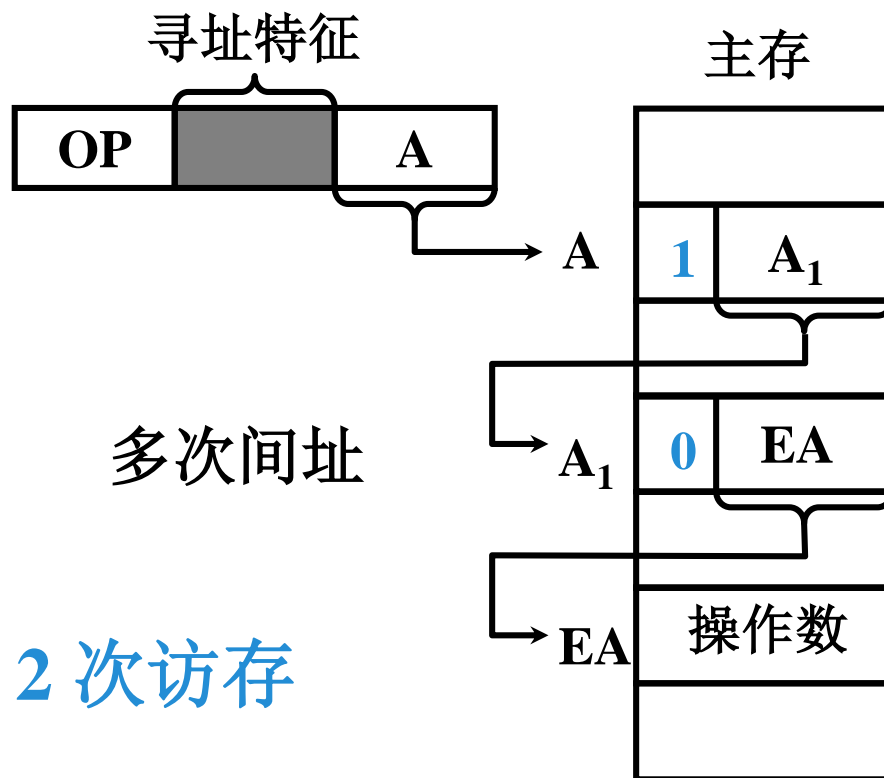
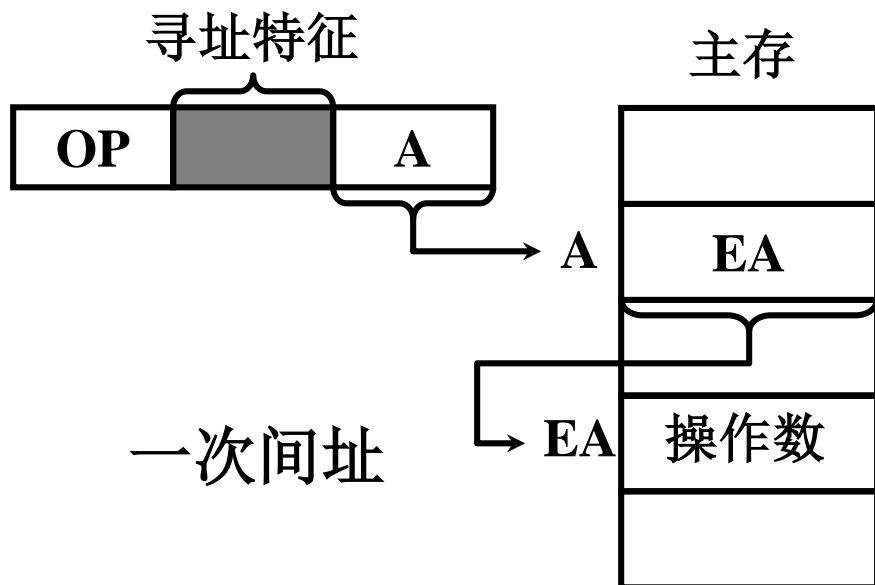
MUL 指令 被乘数隐含在 AX (16位) 或 AL (8位) 中

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

$EA = (A)$  有效地址由形式地址间接提供



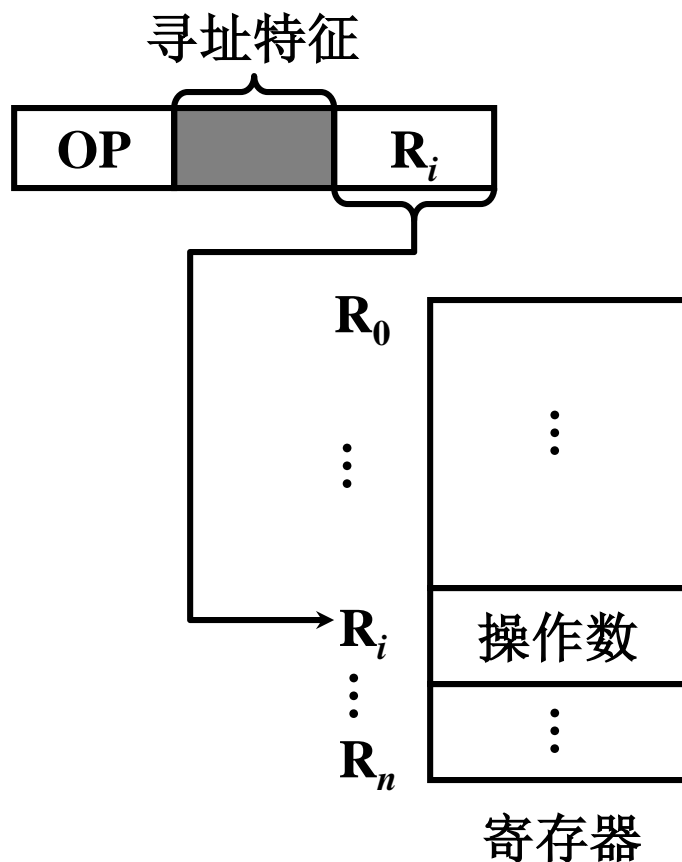
- 执行指令阶段 2 次访存
- 可扩大寻址范围
- 便于编制程序

多次访存





$EA = R_i$       有效地址即为寄存器编号



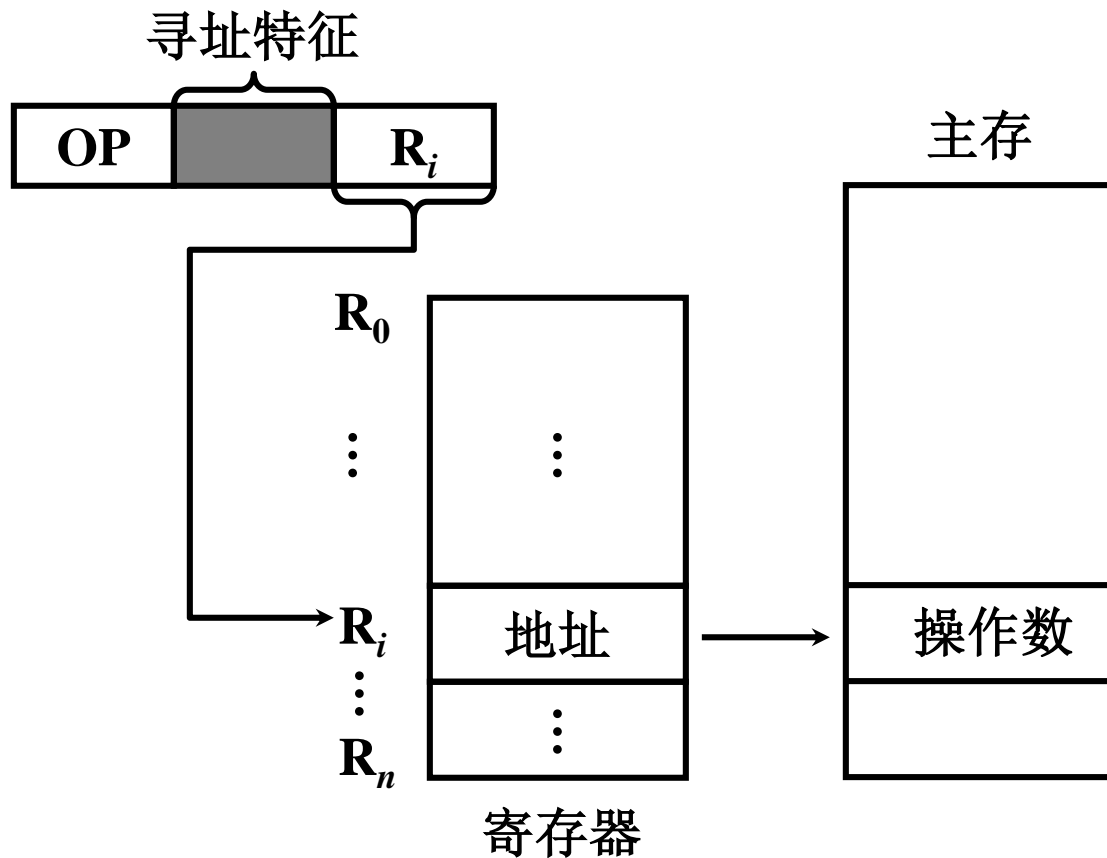
- 执行阶段不访存，只访问寄存器，执行速度快
- 寄存器个数有限，可缩短指令字长

## 6. 寄存器间接寻址

7.3

$EA = (R_i)$

有效地址在寄存器中

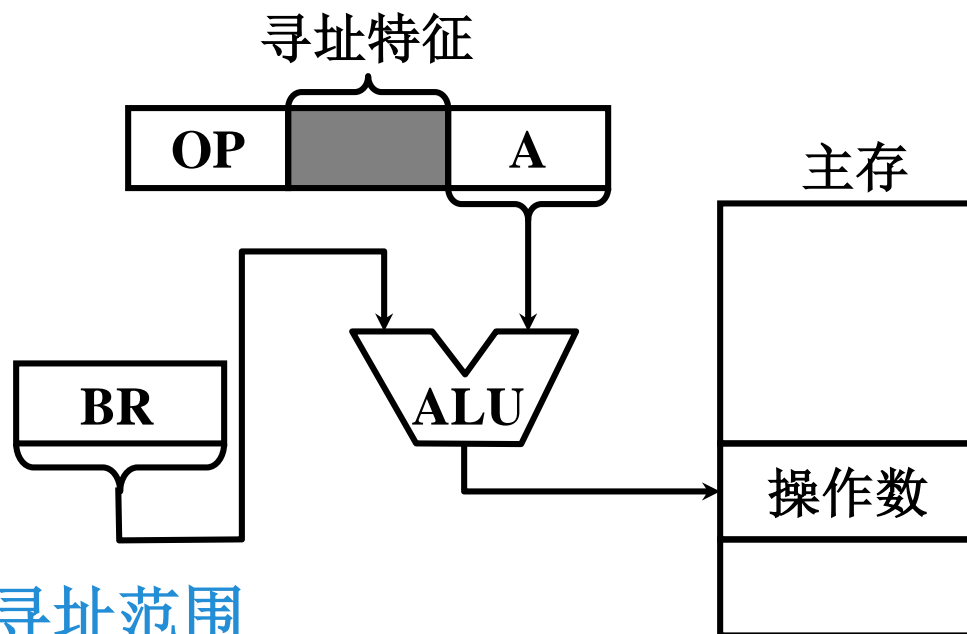


- 有效地址在寄存器中，操作数在存储器中，执行阶段访存
- 便于编制循环程序

### (1) 采用专用寄存器作基址寄存器

$$EA = (BR) + A$$

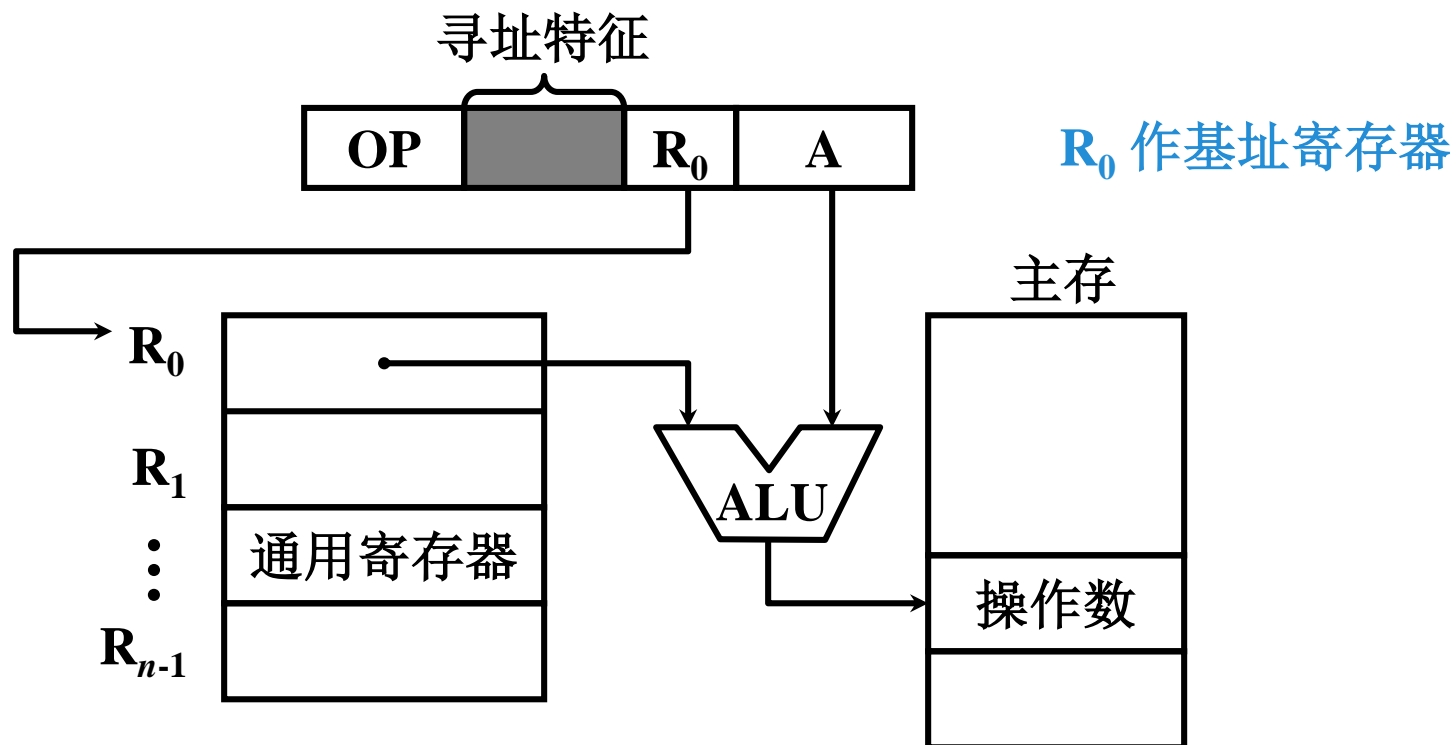
**BR** 为基址寄存器



- 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

7.3



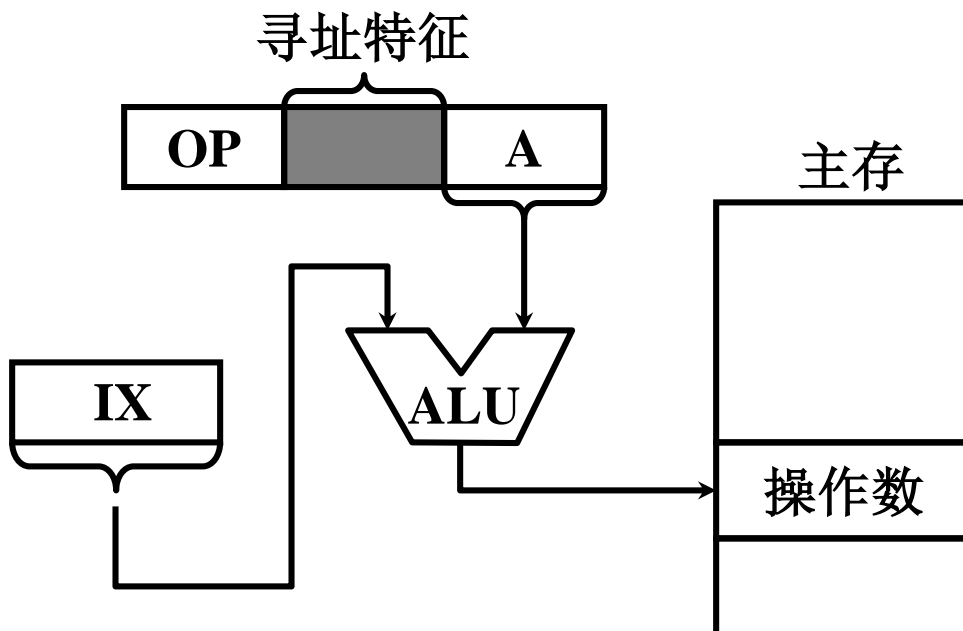
- 由用户指定哪个通用寄存器作为基址寄存器
- 基址寄存器的内容由操作系统确定
- 在程序的执行过程中 **R<sub>0</sub>** 内容不变，形式地址 **A** 可变

# 8. 变址寻址

7.3

$EA = (IX) + A$      $IX$  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**  $D + 2$

$\vdots$

**ADD**  $D + (N - 1)$

**DIV**  $\# N$

**STA**  $ANS$

共  $N + 2$  条指令

## 变址寻址

**LDA**  $\# 0$   $0 \rightarrow ACC$

**LDX**  $\# 0$   $X$  为变址寄存器

**ADD**  $X, D$   $D$  为形式地址

**INX**  $(X) + 1 \rightarrow X$

**CPX**  $\# N$   $(X)$  和  $\# N$  比较

**BNE**  $M$  结果不为零则转

**DIV**  $\# N$

**STA**  $ANS$

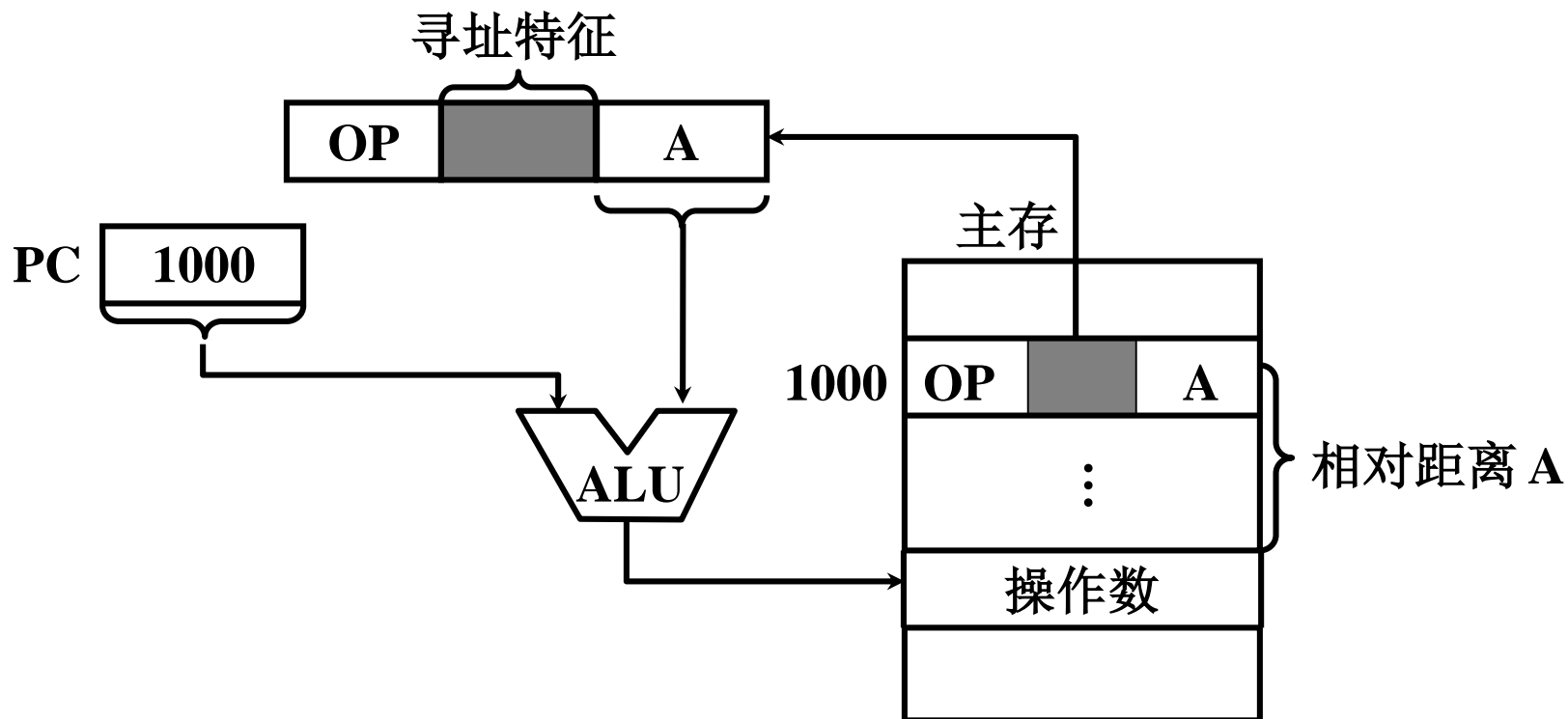
共 8 条指令

## 9. 相对寻址

7.3

$$EA = (PC) + A$$

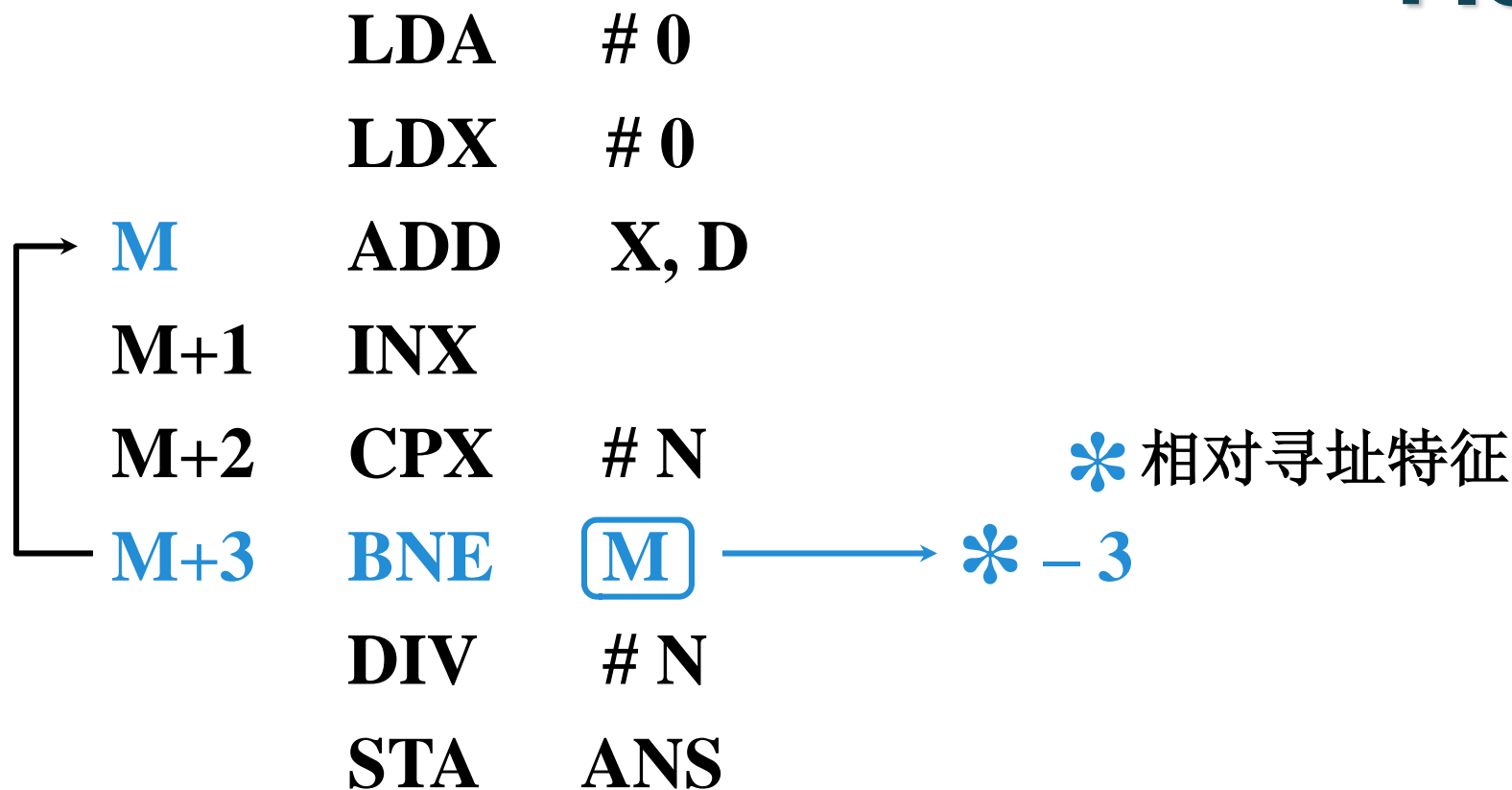
A 是相对于当前指令的位移量（可正可负，补码）



- 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

7.3



**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

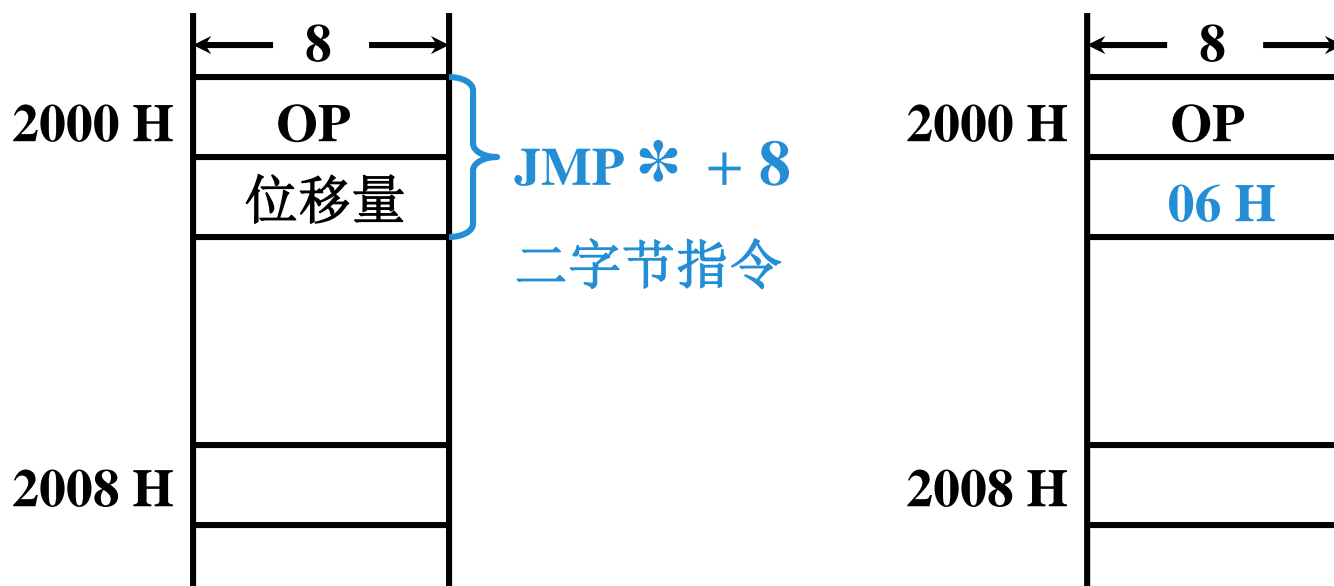
指令 **BNE \* - 3** 操作数的有效地址为

$$EA = (M+3) - 3 = M$$



## (2) 按字节寻址的相对寻址举例

7.3



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

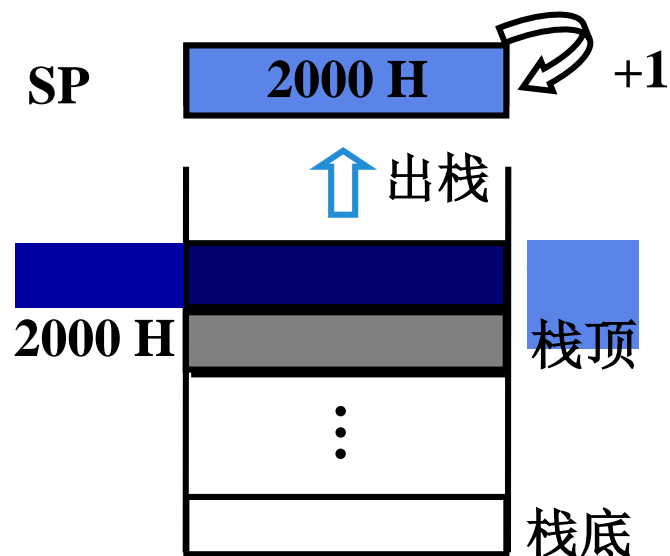
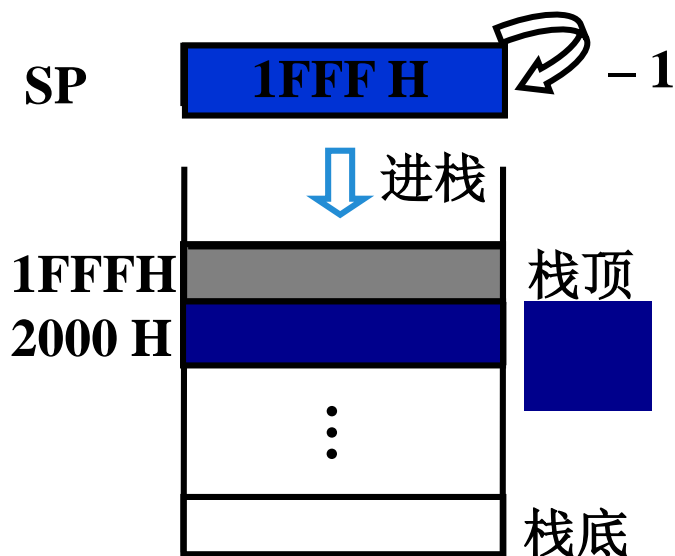
7.3

## (1) 堆栈的特点

堆栈 { 硬堆栈      多个寄存器  
      软堆栈      指定的存储空间

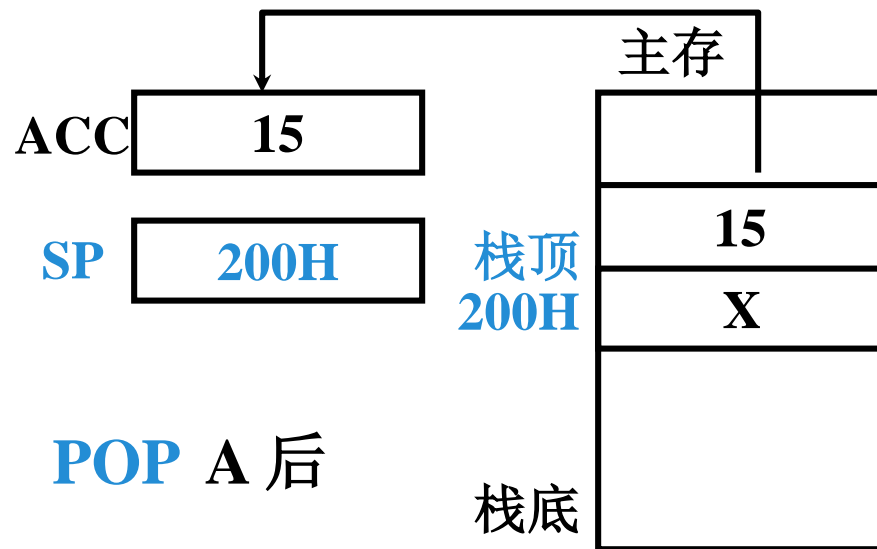
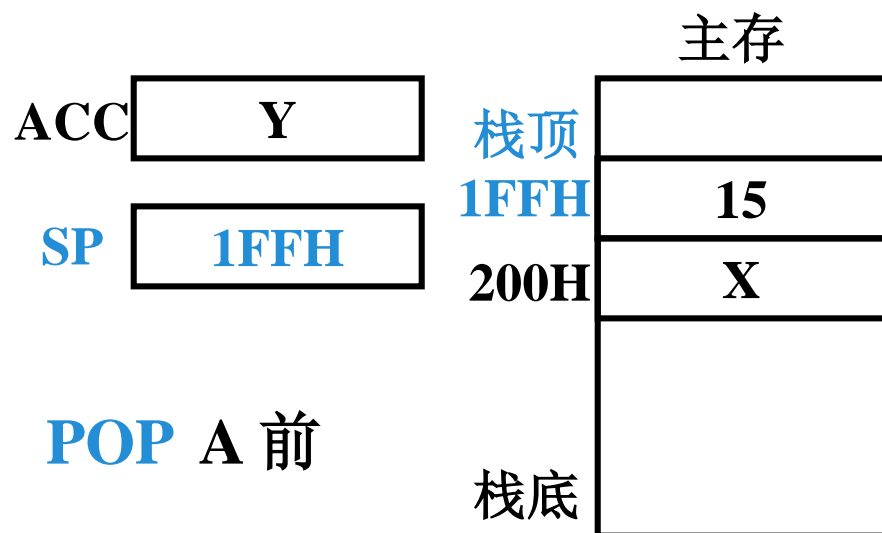
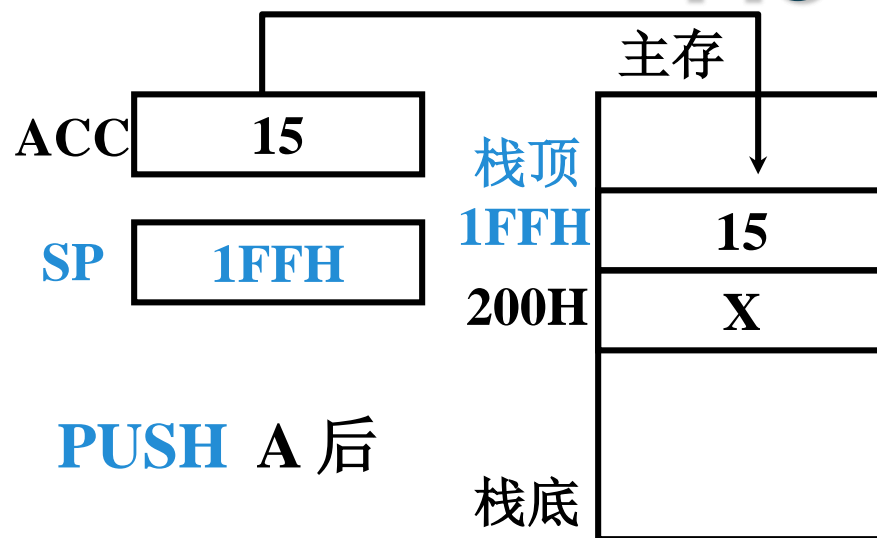
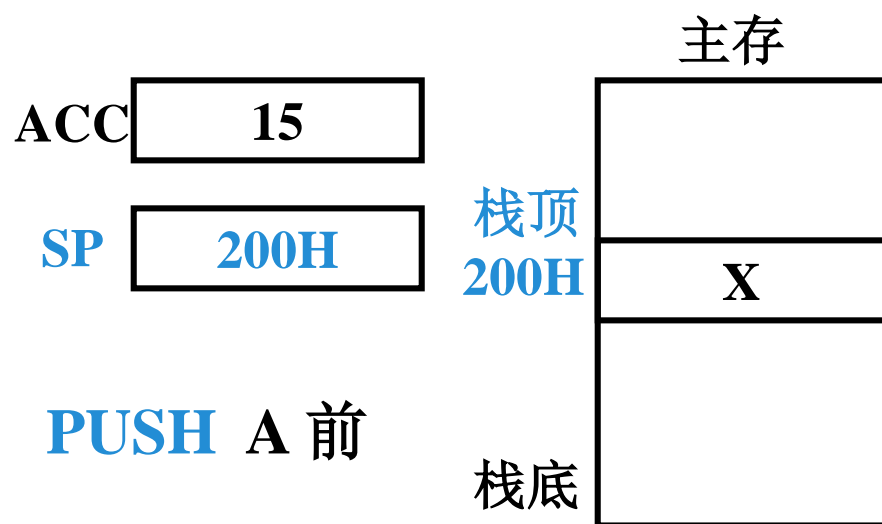
先进后出（一个入出口）    栈顶地址 由 SP 指出

进栈     $(SP) - 1 \rightarrow SP$     出栈     $(SP) + 1 \rightarrow SP$



## (2) 堆栈寻址举例

7.3



### (3) SP 的修改与主存编址方法有关

7.3

#### ① 按 字 编址

进栈  $(SP) - 1 \longrightarrow SP$

出栈  $(SP) + 1 \longrightarrow SP$

#### ② 按 字节 编址

存储字长 16 位 进栈  $(SP) - 2 \longrightarrow SP$

出栈  $(SP) + 2 \longrightarrow SP$

存储字长 32 位 进栈  $(SP) - 4 \longrightarrow SP$

出栈  $(SP) + 4 \longrightarrow SP$

## 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**

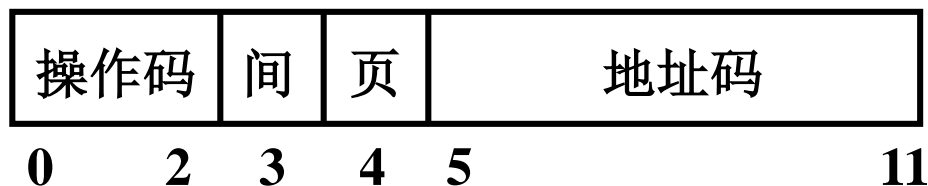
指令寻址、操作数寻址

**寄存器个数**

寄存器的多少直接影响指令的执行时间

### 1. PDP-8 指令字长固定 12 位

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

7.4

指令字长有 16 位、32 位、48 位三种



16

零地址 (16 位)

扩展操作码技术



10

6

一地址 (16 位)

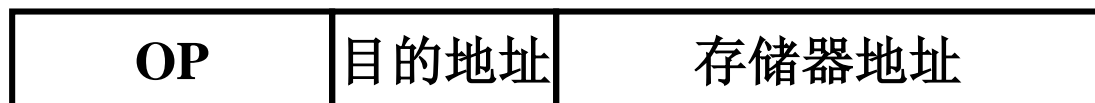


4

6

6

二地址 R – R (16 位)



10

6

16

二地址 R – M (32 位)



4

6

6

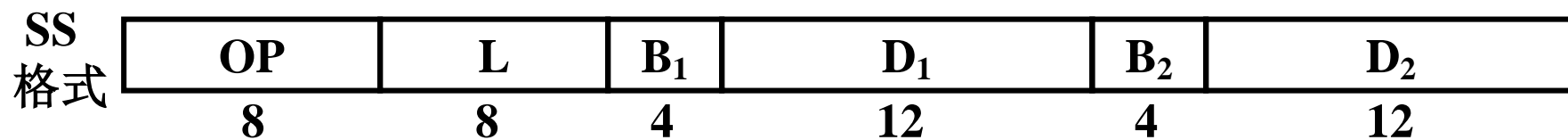
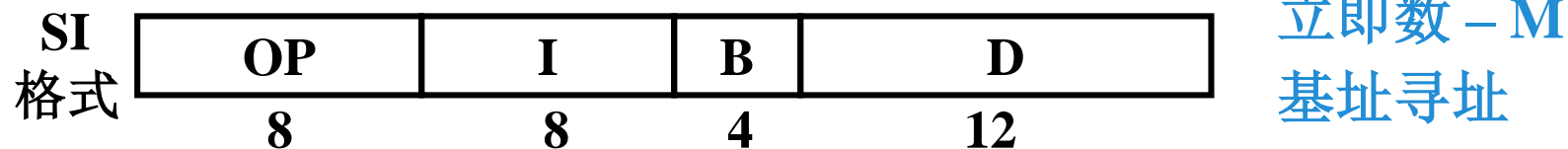
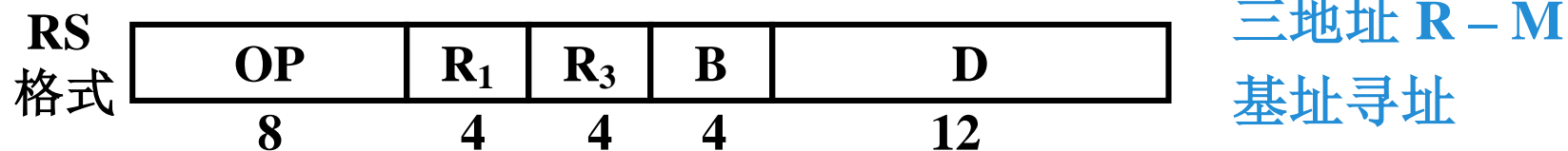
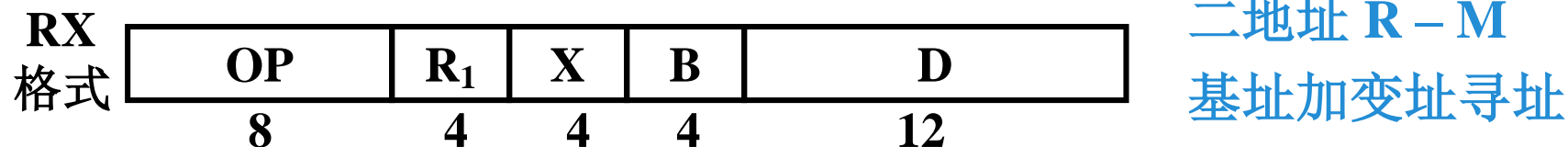
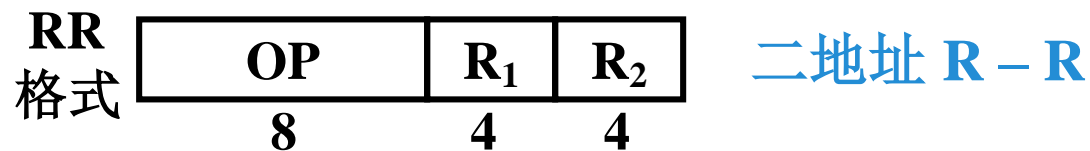
16

16

二地址 M – M (48 位)

# 3. IBM 360

## 7.4



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



### 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



## ❖ 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。

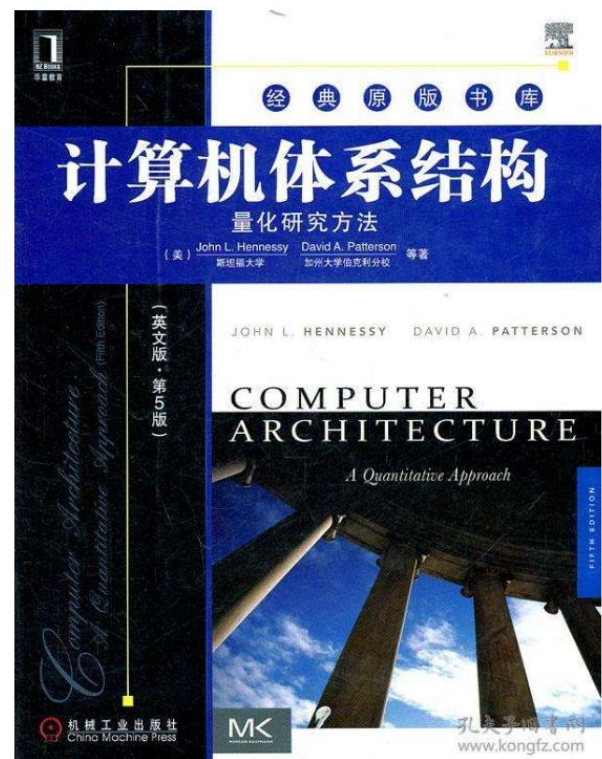
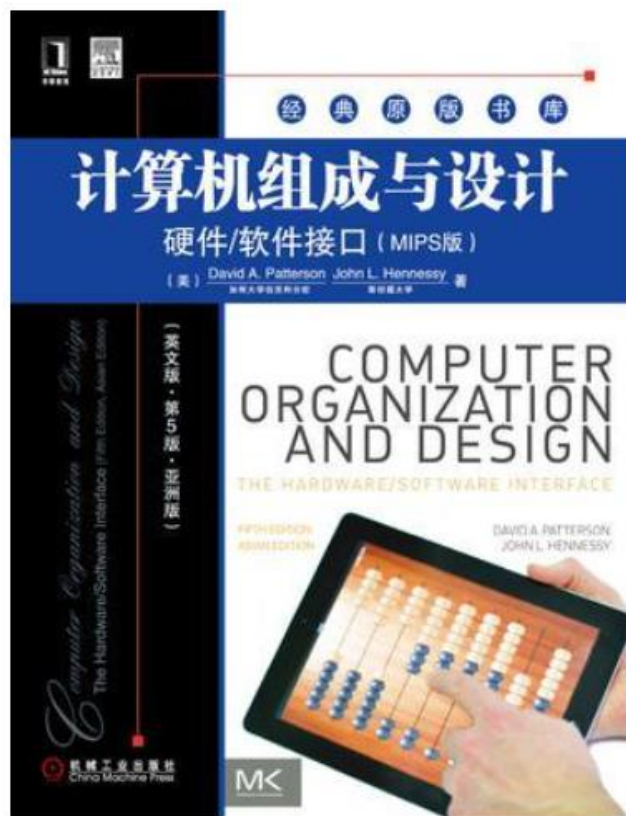
- ❖ The award praised them for "pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry"
- ❖ 开创了计算机体系结构设计和评估的系统的、量化的方法，对微处理器行业具有持久的影响力

# 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。



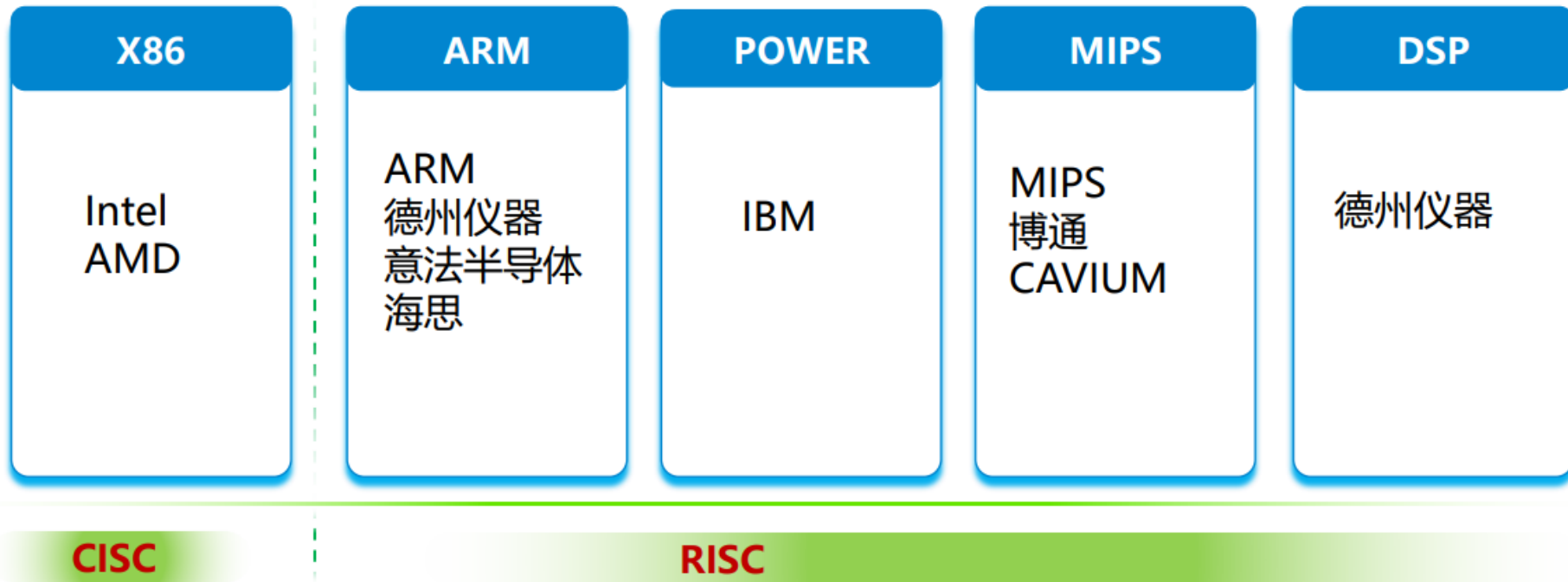
- 选用使用频度较高的一些 简单指令，复杂指令的功能由简单指令来组合
- 指令 长度固定、指令格式种类少、寻址方式少
- 只有 **LOAD / STORE** 指令访存,其余指令的操作都在寄存器之间进行。
- CPU 中有多个 通用 寄存器
- 采用 流水技术，大部分指令在一个时钟周期内完成
- 采用 组合逻辑 实现控制器
- 采用 优化 的 编译 程序

- 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 生成高效的代码

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

## 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

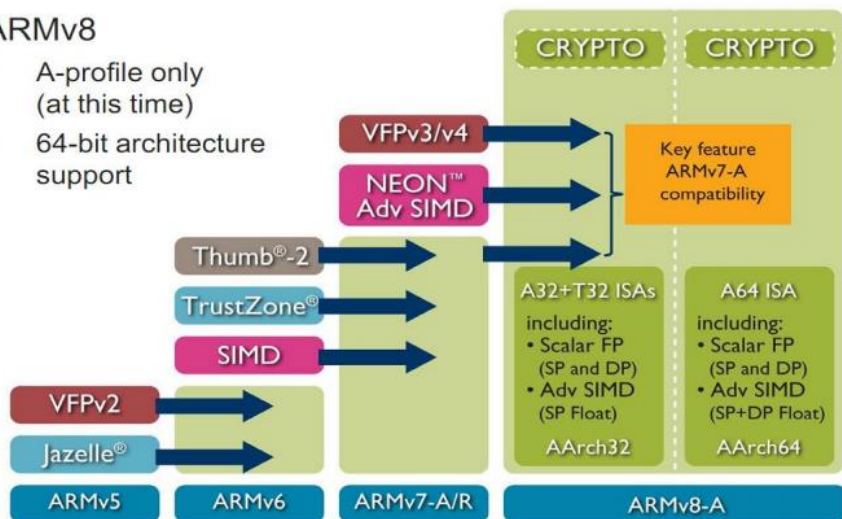
	X86	ARM	POWER
指令集	<ul style="list-style-type: none"><li>• CISC, 复杂指令集</li></ul>	<ul style="list-style-type: none"><li>• RISC, 精简指令集</li></ul>	<ul style="list-style-type: none"><li>• RISC, 精简指令集</li></ul>
架构	<ul style="list-style-type: none"><li>• 重核架构, 高性能高功耗</li></ul>	<ul style="list-style-type: none"><li>• 多核架构, 均衡的性能功耗比</li></ul>	<ul style="list-style-type: none"><li>• 重核架构, 高性能内核</li></ul>
生态	<ul style="list-style-type: none"><li>• 生态非常成熟, 通用性强</li></ul>	<ul style="list-style-type: none"><li>• 生态正在快速发展与完备</li></ul>	<ul style="list-style-type: none"><li>• 聚焦大小型机和HPC</li></ul>
开放性	<ul style="list-style-type: none"><li>• 封闭架构, 英特尔及AMD主导</li></ul>	<ul style="list-style-type: none"><li>• 开放平台, IP授权的商业模式</li></ul>	<ul style="list-style-type: none"><li>• 开放平台 (2019.8), IBM主导</li></ul>

# 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

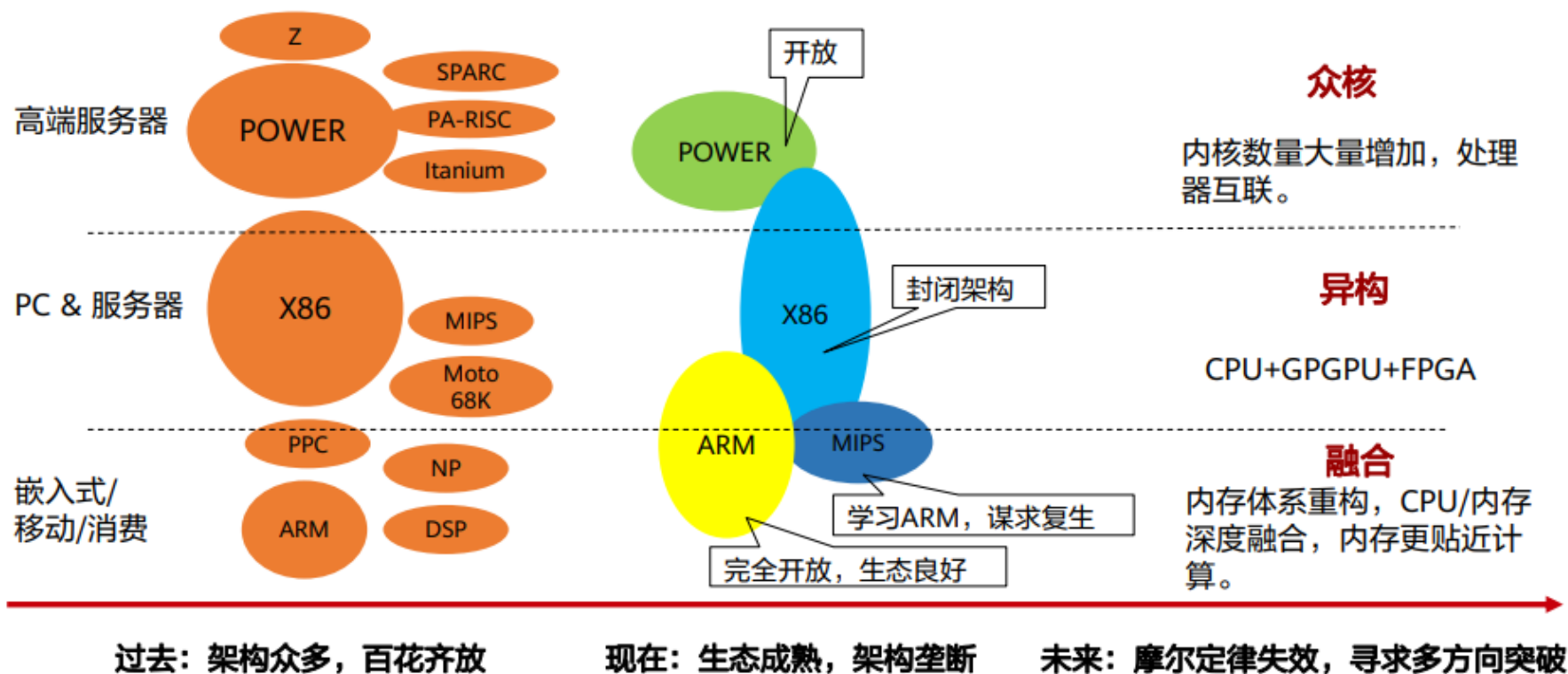
- A-profile only (at this time)
- 64-bit architecture support



## ARMv8架构的特点：

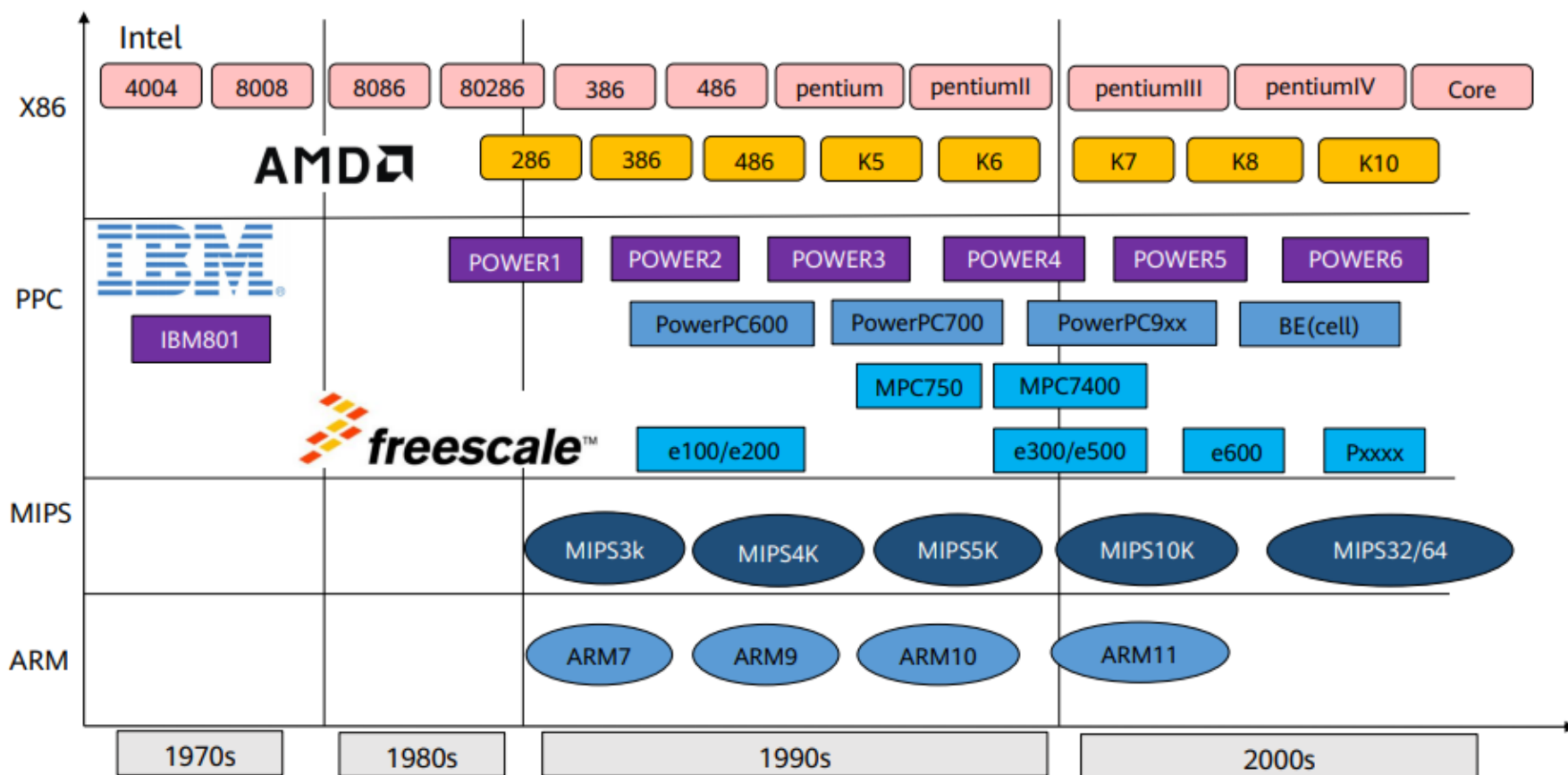
- 31个64位通用寄存器，原来架构只有15个通用寄存器；
- 新指令集支持64位运算，指令中的寄存器编码由4位扩充到5位；
- 新指令集仍然是32位，减少了条件执行指令，条件执行指令的4位编码释放出来用于寄存器编码；
- 堆栈指针SP和程序指针PC都不再是通用寄存器了，同时推出了零值寄存器（类似PowerPC的r0）；
- A64与A32的高级SIMD和FP相同；
- 高级SIMD与VFP共享浮点寄存器，支持128位宽的vector；
- 新增加解密指令。

# 处理器发展趋势





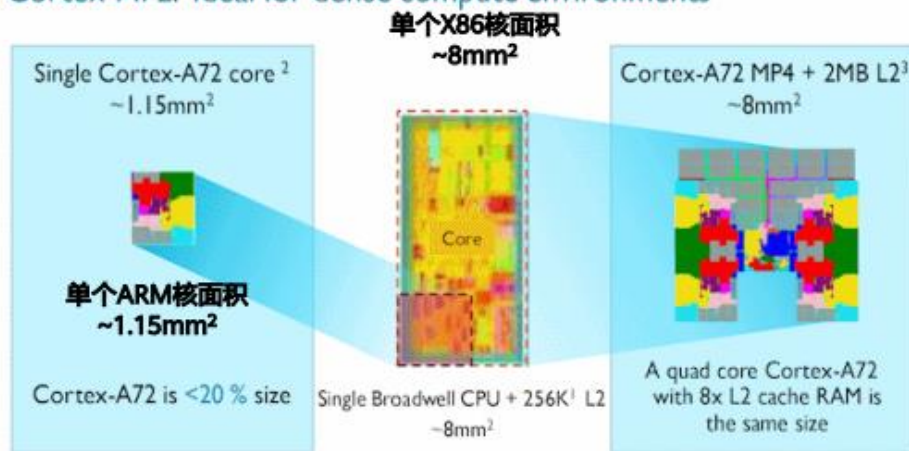
# 主流CPU发展路径




# 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。

Cortex-A72: Ideal for dense compute environments





- 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体系结构的指令集（Instruction Set）

- ARM指令集
- Thumb指令集
- Thumb-2指令集

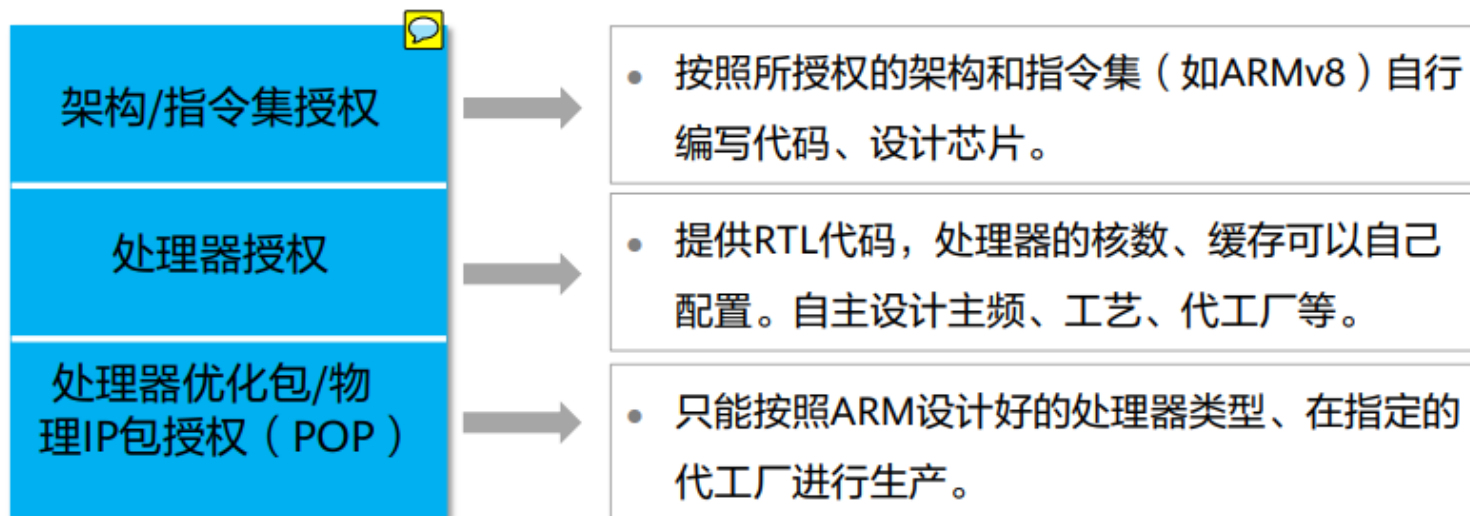


## ❖ ARM的微体系结构（Micro-architecture）

- 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

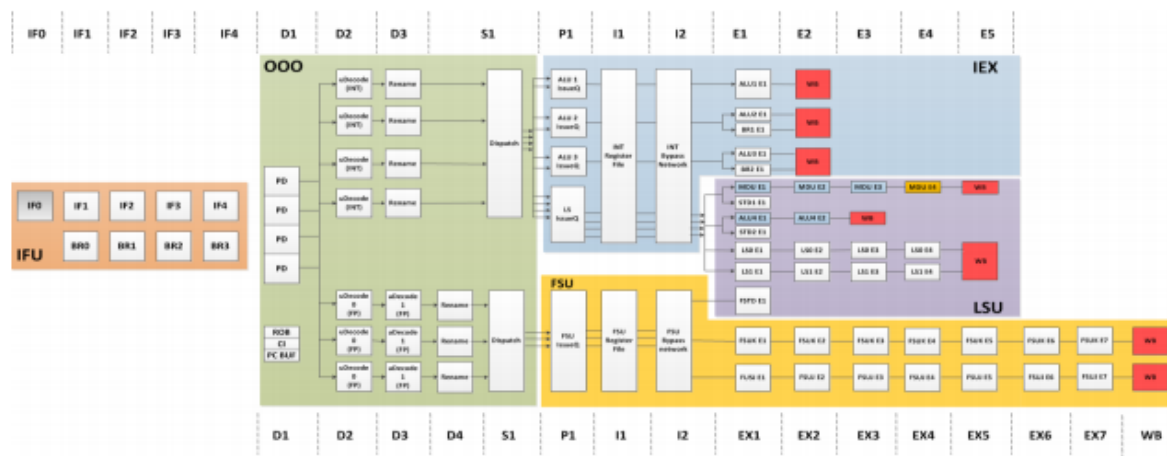
- ARM目前在全球拥有大约1000个授权合作商、320家伙伴，但是购买架构授权的厂家不超过20家，中国有华为、飞腾获得了架构授权。





- 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

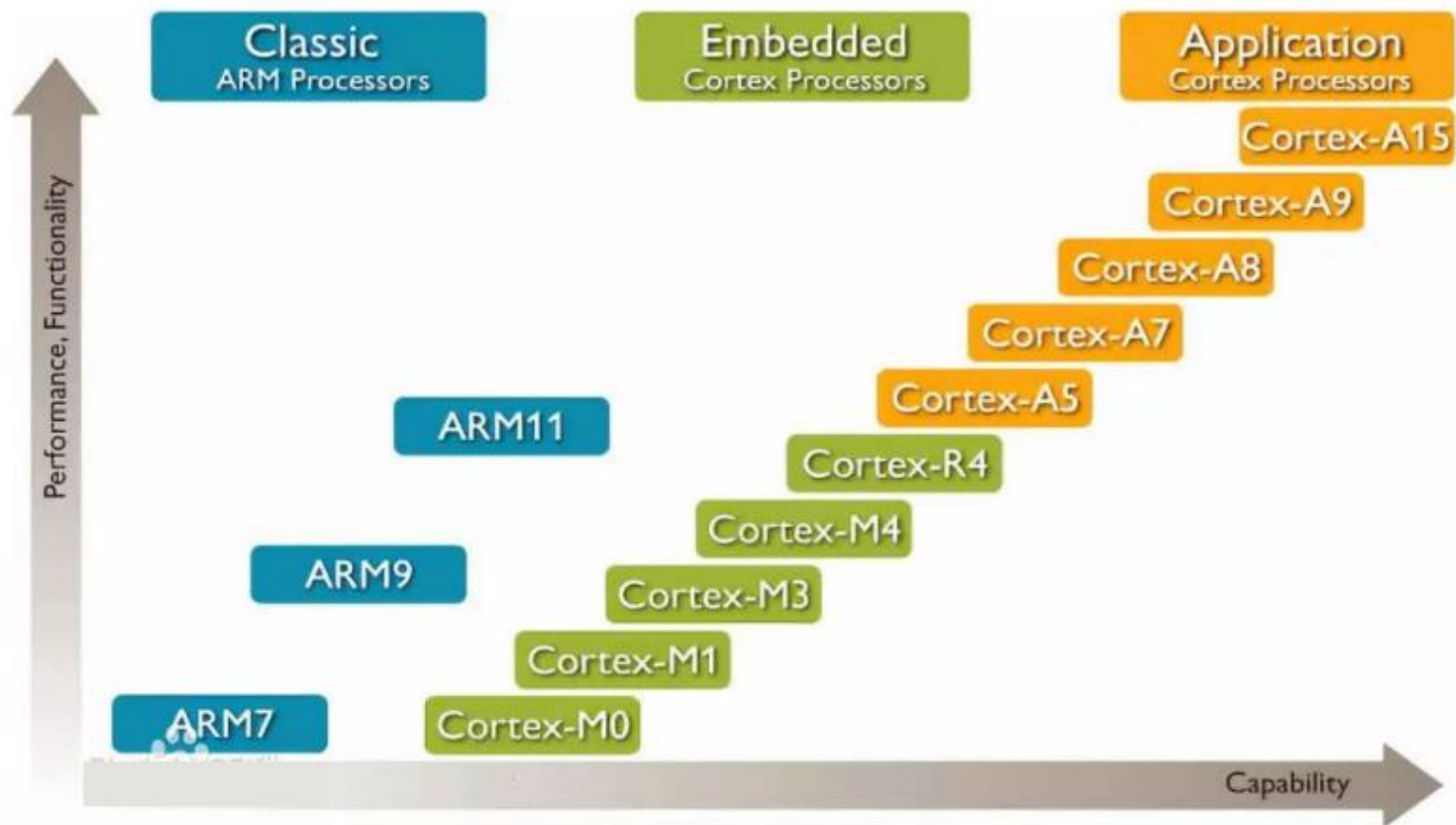
# ARM 处理器系列命名规则

- 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 <sup>[9]</sup>

# ARM架构发展史



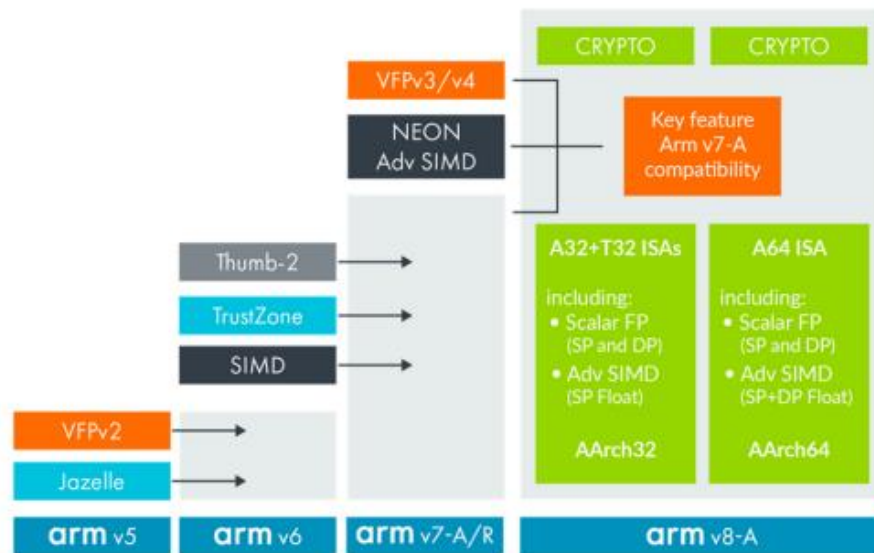
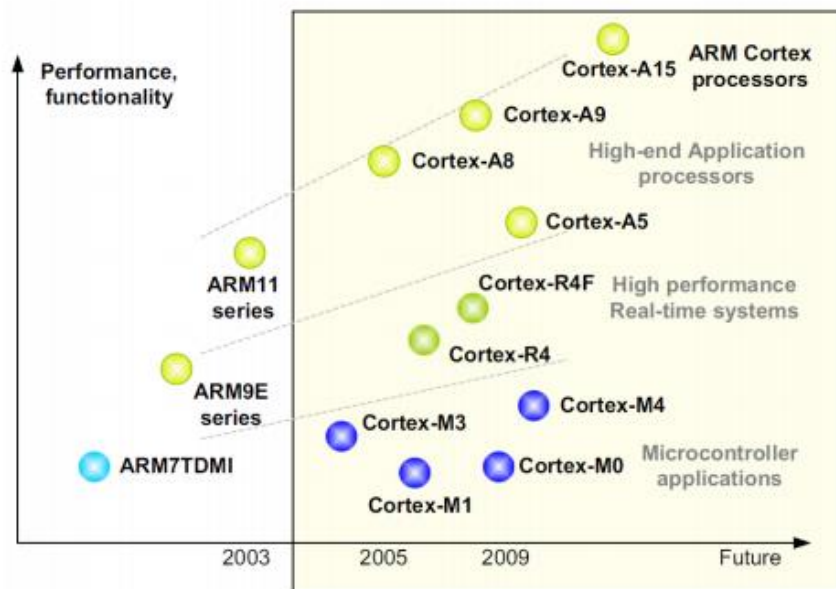
Huawei Confidential





- 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：高效能、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



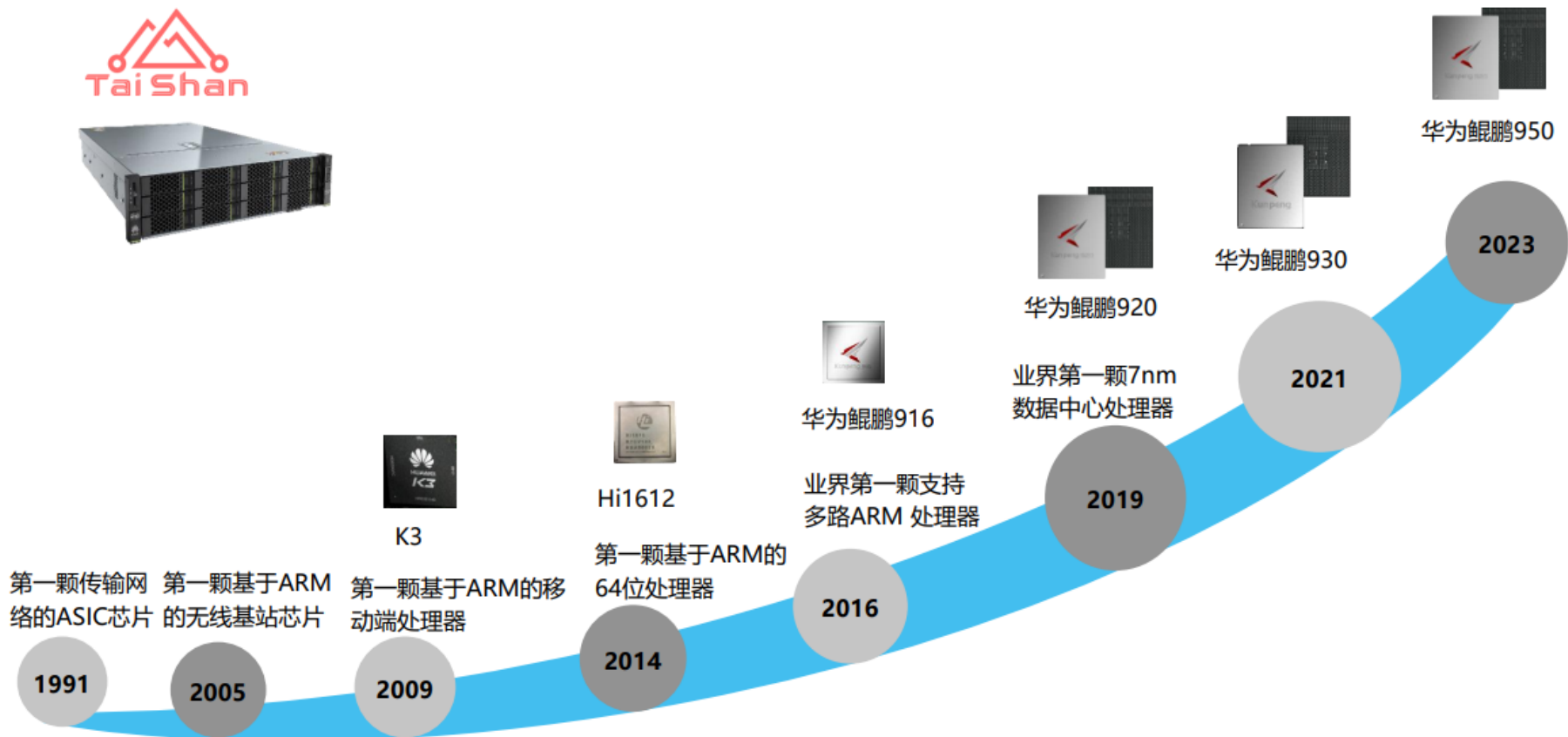
# ARM服务器处理器的优势



- 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

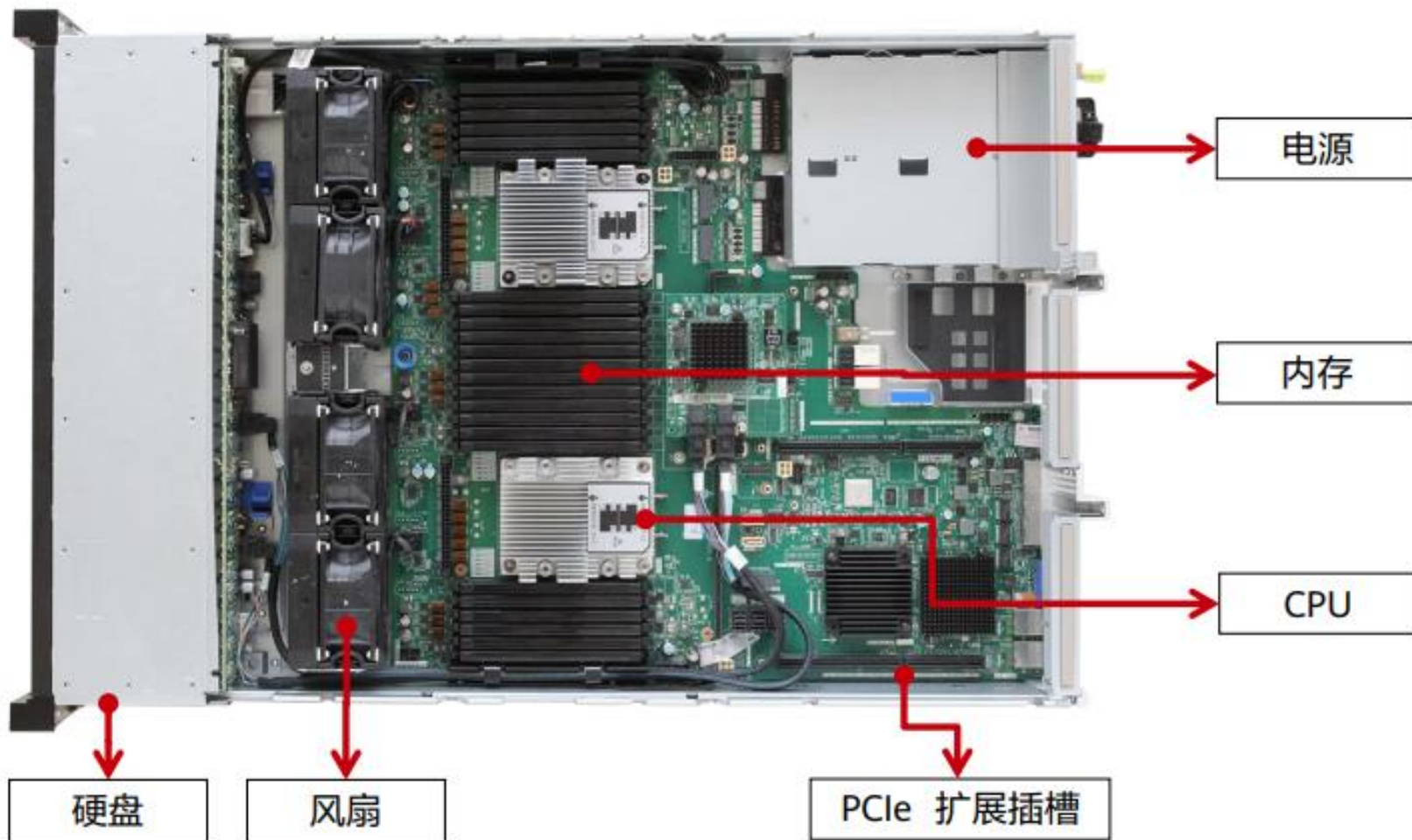


# 华为鲲鹏处理器





# 服务器内部视图



# Thank You !

计算机组成原理

**指令字长为16位，每个地址码为6位，采用扩展操作码的方式，设计14条二地址指令，100条一地址指令，100条零地址指令。**

- (1) 画出扩展图。 (2) 给出指令译码逻辑。  
(3) 计算操作码平均长度。

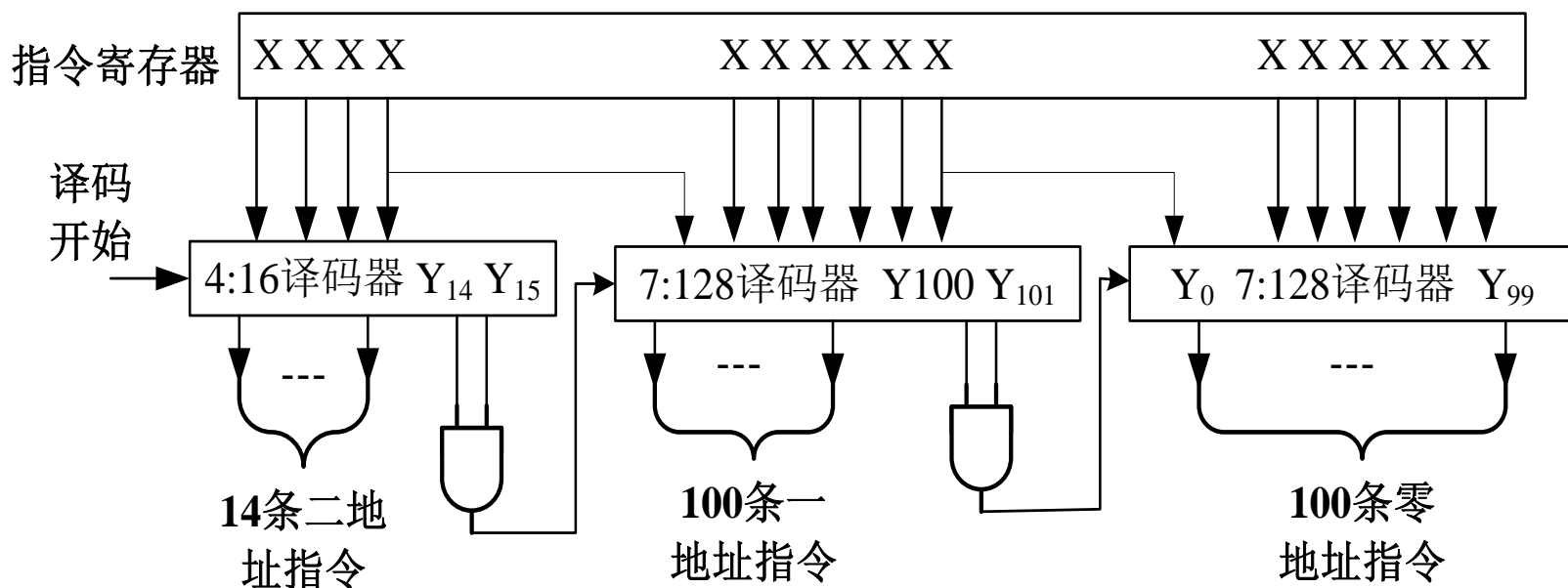
**解： (1) 操作码扩展如下：**

0 0 0 0	x x x x x x	x x x x x x	14条二地址指令
⋮	⋮	⋮	
1 1 0 1	x x x x x x	x x x x x x	100条二地址指令
1 1 1 0	0 0 0 0 0 0	x x x x x x	
⋮	⋮	⋮	
1 1 1 1	1 0 0 0 1 1	x x x x x x	100条二地址指令
1 1 1 1	1 0 0 1 0 0	0 0 0 0 0 0	
⋮	⋮	⋮	
1 1 1 1	1 0 0 1 0 1	1 0 0 0 1 1	

**指令字长为16位，每个地址码为6位，采用扩展操作码的方式，设计14条二地址指令，100条一地址指令，100条零地址指令。**

- (1) 画出扩展图。 (2) 给出指令译码逻辑。  
(3) 计算操作码平均长度。

**解： (2) 指令译码逻辑：**



**指令字长为16位，每个地址码为6位，采用扩展操作码的方式，设计14条二地址指令，100条一地址指令，100条零地址指令。**

- (1) 画出扩展图。 (2) 给出指令译码逻辑。**
- (3) 计算操作码平均长度。**

**解： (3) 操作码平均长度**

$$= (4 \times 14 + 10 \times 100 + 16 \times 100) / 214 \approx 12.4$$

例：设相对寻址的转移指令占**两个字节**，第一字节是操作码，第二字节是相对位移量，用**补码表示**。每当CPU从存储器取出一个字节时，即自动完成 $(PC)+1 \rightarrow PC$ 。

(1) 设当前PC值为3000H，问转移后的目标地址范围是多少？

解：(1) 由于相对寻址的转移指令为两个字节，第一个字节为操作码，第二个字节为相对位移量，且用补码表示，故其范围为-128 ~ +127，即80H ~ 7FH。又因PC当前值为3000H，且CPU取出该指令后，PC已修改为3002H，因此最终的转移目标地址范围为3081H ~ 2F82H，即 $3002H + 7FH = 3081H$ 至 $3002H - 80H = 2F82H$

思考：若PC为16位，位移量可正可负，PC相对寻址范围为多大？

解：相对寻址中，PC提供基准地址，位移量提供修改量，位移量为16位可正可负，则相对寻址范围为： $(PC) - 2^{15} \sim (PC) + 2^{15} - 1$

**例：设相对寻址的转移指令占两个字节，第一字节是操作码，第二字节是相对位移量，用补码表示。每当CPU从存储器取出一个字节时，即自动完成 $(PC)+1 \rightarrow PC$ 。**

**(2) 若当前PC值为2000H,要求转移到201BH，则转移指令第二字节的内容是什么？**

**解： (2) 若PC当前值为2000H，取出该指令后PC值为2002H，故转移指令第二字节应为**  
 **$201BH - 002H = 19H$ 。**

**例：设相对寻址的转移指令占两个字节，第一字节是操作码，第二字节是相对位移量，用补码表示。每当CPU从存储器取出一个字节时，即自动完成 $(PC)+1 \rightarrow PC$ 。**

**(3) 若当前PC值为2000H，指令JMP \* -9 (\*为相对寻址特征) 的第二字节的内容是什么？**

**解：根据汇编语言指令JMP\* - 9，即要求转移后的目标地址为 $2000H - 09H = 1FF7H$ ，但因为CPU取出该指令后PC值已修改为2002H，故转移指令的第二字节的内容应为-11（十进制），写成补码为F5H。**