中国科学技术大学
University of Science and Technology of China

第二章 指令系统

王 超

中国科学技术大学计算机学院
高能效智能计算实验室

2026年春

主要内容

1. 指令系统概述
   - 1.1 指令系统的发展
   - 1.2 指令系统的性能要求
   - 1.3 编程语言与硬件的关系

2. 指令格式
   - 2.1 指令的一般格式
   - 2.2 指令字长
   - 2.3 指令助记符

3. 操作数与操作类型
   - 3.1 操作数类型
   - 3.2 数据在存储器中的存储方式
   - 3.3 操作类型

4. 寻址方式
   - 4.1 指令寻址
   - 4.2 数据寻址

5. CISC与RISC
   - 6.1 CISC技术
   - 6.2 RISC技术
   - 6.3 CISC与RISC的比较

6. 指令系统设计与举例
   - 5.1 典型指令
   - 5.2 典型指令系统
   - 5.3 指令系统设计及举例

1.1 指令系统的发展

□指令

- ✓计算机执行某种操作的命令
- ✓从计算机组成的层次结构看，指令分为
  - 微指令：微程序级的命令，属于硬件
  - 宏指令：若干条机器指令组成的命令，属于软件
  - 机器指令：通常所说的指令，介于微指令和宏指令之前，可完成一个独立的算术运算或逻辑运算操作

□指令系统

- ✓一台计算机所有机器指令的集合，称为该计算机的指令系统
- ✓表征计算机性能的重要因素
  - 指令的格式和功能直接影响机器的硬件结构，也直接影响系统软件，影响到机器的适用范围

1.1 指令系统的发展（2）

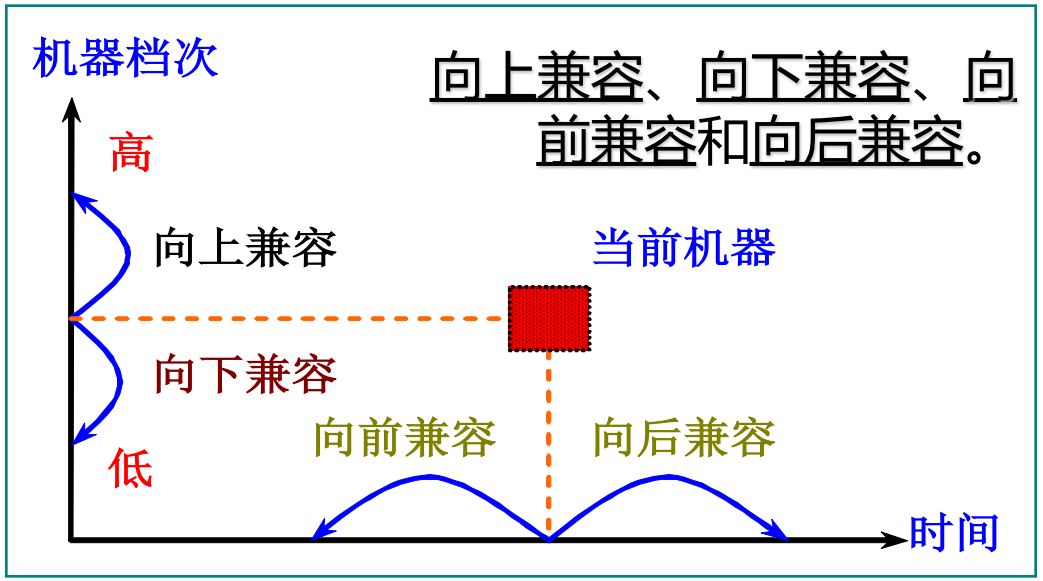
□ 系列计算机

- ✓ 基本指令系统相同、基本体系结构相同的一系列计算机
  - 不同型号，推出时间不同、器件不同，结构和性能上有差异
- ✓ 解决软件兼容的问题——新机器的指令系统包含旧机器的所有指令

计算机	时间	处理器	字宽	主要I/O总线	存储空间
PC和PC XT	1981	8088	16位	PC总线	20位
PC AT	1982	80286	16位	AT（ISA）	24位
80386 PC	1985	80386	32位	ISA/EISA	32位
80486 PC	1989	80486	32位	ISA+VL	32位
Pentium PC	1993	Pentium	32位	ISA+PCI	32位
Pentium II PC	1997	Pentium II	32位	ISA+PCI+AGP	32位
Pentium III PC	1999	Pentium III	32位	PCI+AGP +USB	32位
Pentium 4 PC	2000	Pentium 4	32位	PCI-X+AGP +USB	32位

兼容性

- 向上(下)兼容指的是按某档机器编制的程序，不加修改的就能运行于比它高(低)档的机器。



- 向前(后)兼容指的是按某个时期投入市场的某种型号机器编制的程序，不加修改地就能运行于在它之前(后)投入市场的机器。
- 性能和成本的权衡

1.1 指令系统的发展（3）

口指令系统的复杂化和精简

- ✓复杂指令集计算机CISC
  - VLSI技术的发展，计算机硬件结构复杂化，指令系统庞大
  - “二八定律”
- ✓精简指令集计算机RISC

为什么会出现CISC到RISC的转变?

1.1 指令系统的发展（3）

执行频率排序	80X86指令	指令执行频率
1	Load	22%
2	条件分支	20%
3	比较	16%
4	Store	12%
5	加	8%
6	与	6%
7	减	5%
8	寄存器－寄存器间数据移动	4%
9	调用	1%
10	返回	1%
合　计		96%

1.2 指令系统的性能要求

- 指令系统的设计是计算机系统设计的核心问题
  - ✓ 指令系统的性能，决定了计算机的基本功能
  - ✓ 与计算机硬件密切相关，也直接关系到用户的使用需要
- 完善的指令系统的四方面要求
  - ✓ 完备性
    - 指令系统直接提供的指令足够使用，相关功能不需用软件实现
      (乘法指令 V.S.加法指令)
  - ✓ 有效性
    - 指令级程序占据的存储空间小，执行速度快，能够高效运行
      (访存效率+执行效率)
  - ✓ 规整性
    - 对称性、匀齐性、指令格式与数据格式的一致性
      (变长指令V.S.定长指令)
  - ✓ 兼容性
    - 向上兼容—低档机器的软件能在高档机器上运行、向前兼容

1.3 编程语言与硬件的关系

计算机语言

- ✓ 程序设计中使用的工具，分为高级语言和低级语言
  - 高级语言：如C、C++、JAVA等，语句和用法与具体机器的指令系统无关
  - 低级语言：机器语言/汇编语言，和具体机器的指令系统密切相关
- ✓ 计算机只能识别和执行机器语言程序，需借助汇编器、编译器或解释器，将汇编语言或高级语言翻译为机器语言

高级语言与低级语言的比较

- ✓ 高级语言在编程方面更便利和优越，但其不知硬件结构，性能上相对低级语言较差
- ✓ 高级语言不能编写直接访问机器硬件资源的系统软件或设备驱动
  - 一些高级语言，如C语言，提供了与汇编语言之间的调用接口

常见的语言泛型

- 过程式范型（Procedural，或称命令式，Imperative）：
  - ✓ C，Pascal，Fortran 等
- 函数式范型（Functional）：
  - ✓ 最早的函数式语言是John McCarthy 开发的Lisp。目前最重要的函数式语言包括Common Lisp，Scheme，ML 和Haskell 等。
- 面向对象的范型（Object-Oriented）
  - ✓ Smalltalk，Java 等基于类的语言，程序中定义类，对象是类的实例
  - ✓ JavaScript，是基于对象的语言，通过原型概念定义新的类似对象
  - ✓ 其他支持OO 概念和编程方法的语言包括C++，Ada95 等
- 说明式语言（Declarative）
  - ✓ 逻辑式语言Prolog；数据库查询语言Datalog
- 脚本语言（Scripting Language）
- 文本描述语言 Postscript，也是一种完整的程序设计语言
- 仪器控制语言 Forth，是一种完整的基于栈的程序设计语言
- 硬件描述语言 Verilog 和 VHDL … …

编程语言与硬件的关系

	比较内容	高级语言	低级语言
1	对程序员的训练要求 (1)通用算法 (2)语言规则 (3)硬件知识	有 较少 不要	有 较多 要
2	对机器独立的程度	独立	不独立
3	编制程序的难易程度	易	难
4	编制程序所需时间	短	较长
5	程序执行时间	较长	短
6	编译过程中对计算机资源的要求	多	少

主要内容

1. 指令系统概述
   - 1.1 指令系统的发展
   - 1.2 指令系统的性能要求
   - 1.3 编程语言与硬件的关系
2. 指令格式
   - 2.1 指令的一般格式
   - 2.2 指令字长
   - 2.3 指令助记符
3. 操作数与操作类型
   - 3.1 操作数类型
   - 3.2 数据在存储器中的存储方式
   - 3.3 操作类型
4. 寻址方式
   - 4.1 指令寻址
   - 4.2 数据寻址
5. CISC与RISC
   - 5.1 CISC技术
   - 5.2 RISC技术
   - 5.3 CISC与RISC的比较
6. 指令系统设计与举例
   - 6.1 典型指令
   - 6.2 典型指令系统
   - 6.3 指令系统设计及举例

2.1 指令的一般格式

□指令格式

- ✓指令字用二进制代码表示的结构形式
- ✓通常由操作码和地址码两部分组成
  - 操作码：指明指令要完成的操作特性和功能
  - 地址码（操作数）：指明参与操作的操作数的存储位置

操作码字段OP	地址码字段A

操作码

操作码作用

- ✓ 指明指令要完成的操作，如加法、传送、移位等
- ✓ 通过操作码字段的不同编码实现

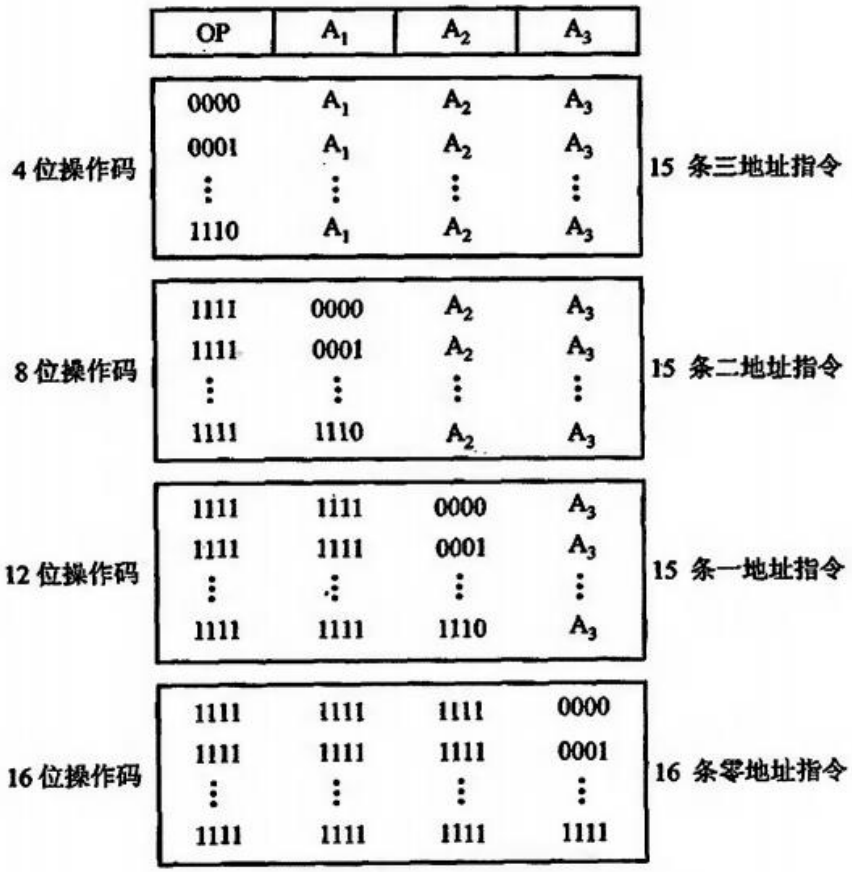
操作码长度

- ✓ 操作码的位数反映了机器的操作种类，即机器最多允许的指令种类数
  - 操作码包含n位的机器，最多能够有2^n种指令。如操作码占7位，则该机器最多包含$2^7=128$种指令
- ✓ 操作码长度可变

	形式	优缺点	应用
固定长度	操作码集中在指令字的某一字段	便于硬件设计，指令译码时间短	大中型计算机，RISC
可变长度	操作码分散在指令字的不同字段	有效压缩操作码平均长度，指令译码分析难度增加	单片机，Intel X86

操作码（2）

- 扩展操作码技术
  - ✓操作码长度随指令字中地址数的变化而变化
  - ✓尽可能以较短的指令字长表示较多的操作种类
- 变长操作码设计原则
  - ✓尽量安排使用频度较高的指令占用短的操作码，使用频度低的指令占用长的操作码，以加速常用指令的译码分析

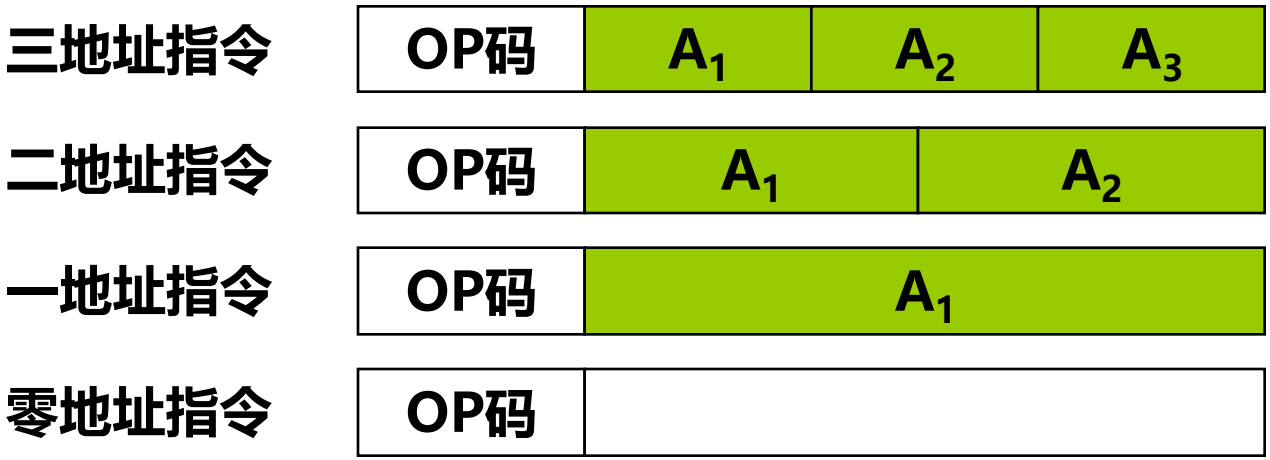


固定4位操作码，能表示的指令种数$2^4=16$种 → 扩展操作码，指令种数61种

地址码

口地址码

- ✓指出指令的**源操作数**的地址（一个或两个）、**结果的地址或者下一条指令的地址**
  - 这里的“地址”，可以是**主存地址、寄存器地址、I/O设备地址**等
- ✓按地址个数分类指令



地址码（2）

- 三地址指令
  - ✓ 最典型的指令格式
  - ✓ 操作：(A_1) op $(A_2) \rightarrow A_3$
  - ✓ 若指令字长32位，操作码8位，地址字段各8位，则三地址指令操作数的直接寻址范围为$2^8=256$
  - ✓ 若地址字段都为主存地址，则完成一条三地址指令操作，共需访问4次主存
    - 取指令1次，取两个操作数2次，存放结果1次

- 二地址指令
  - ✓ 将指令的操作结果保存到某个操作数的地址位置
  - ✓ 操作：(A_1) op $(A_2) \rightarrow A_1$或A_2
    - 可以将结果暂存在CPU的累加器ACC中，(A_1) op $(A_2) \rightarrow$ ACC
  - ✓ 4次或3次访存

地址码（3）

- 一地址指令
  - ✓把一个操作数隐含在累加器ACC中，操作结果又放回ACC中
  - ✓操作： (A_1) op (ACC) → ACC
  - ✓只需2次访存

- 零地址指令
  - ✓指令字中只有操作码，没有地址码
  - ✓零地址指令示例
    - 空操作 NOP
    - 停机指令 HLT
    - 函数返回指令 RET、中断返回指令IRET

指令格式示例（唐书7.4）

例3.1 假设指令字长为16位，操作数的地址码为6位，指令分零地址、一地址、二地址三种格式。

1）设操作码固定，若零地址指令有P种，一地址指令有Q种，问二地址指令最多有几种?

2）采用扩展操作码技术，若二地址指令有X种，零地址指令最多有Y种，则一地址指令有几种?

解：

1）操作数地址码6位，则二地址指令中操作码的位数为16-6-6=4位，则操作码（即指令种类）可有2^4=16种。因为操作码固定，所以二地址指令最多有16-P-Q种

2）采用扩展操作码技术，二地址、一地址、零地址操作码长度分别为4、10、16位。可见，二地址指令操作码每减少一种，就可多构成$2^{10-4=6}$种一地址指令；一地址指令操作码每减少一种，就可多构成$2^{16-10=6}$种零地址指令。

二地址指令有X种，则一地址指令最多有$(2^4-X)\times2^6$种。设实际一地址指令有M种，则零地址指令最多有$[(2^4-X)\times2^6-M])\times2^6$种，即：

$$Y=[(2^4-X)\times2^6-M])\times2^6$$

则一地址指令

$$M=(2^4-X)\times2^6 - Y\times2^{-6}$$

按操作数物理位置的指令分类

□按操作数物理位置的指令分类

- ✓存储器-存储器（SS）型指令
  - 操作数都存放在主存中，指令执行过程需多次访问主存
- ✓寄存器-寄存器（RR）型指令
  - 操作数都存放在CPU的寄存器中，指令执行过程需使用到多个通用寄存器或个别专用寄存器
  - 执行速度相对较快
  - RISC型指令系统中，多数指令都设计为此种类型
- ✓寄存器-存储器（RS）型指令
  - 操作数可能存放在寄存器或存储器中

在计算机的具体设计实现，特别是CISC型指令系统计算机中，指令字长度和指令中的地址结构常不是单一的，往往采用各种格式的混合使用

2.2 指令字长

- 指令字长
  - ✓一个指令字中包含的二进制代码的位数
  - ✓取决于操作码长度、操作数地址的长度和操作数地址的个数
- 指令字长与机器字长的关系
  - ✓指令字长等于机器字长时——**单字长指令**
    - 早期机器中，存储字长也与两者相等，一次访存可以取出一条完整指令或数据
  - ✓指令字长等于机器字长的一半——**半字长指令**
  - ✓指令字长等于机器字长的两倍/多倍——**双/多字长指令**
    - 为了能够提供足够的地址位以解决访问内存任意单元的寻址问题
    - 需多次内存访问才能取出一条完整指令，CPU速度降低，占用更多存储空间

指令字长（2）

口 按指令字长是否可变划分

- ✓ 等长指令结构——指令系统中的所有指令字的长度都相等
  - 可以都为单字长指令或半字长指令等
  - 结构简单，控制方便
- ✓ 变长指令结构——指令系统中的各种指令字的长度因功能而异，可以采用不同位数
  - 结构灵活，充分利用指令长度，但控制较复杂
  - 为提高指令运行速度和节省存储空间，设计指令系统时，通常尽可能把常用的指令设计为单字长或半字长格式

DEC PDP-8的指令字长固定为12位

IBM 370指令字长可为16位(半字) / 32位(单字) / 48位(一字半)

Intel X86系列的指令字长可为8/16/24/32/40/48/64位

一般指令字长都取8的整数倍

2.3 指令助记符

口指令助记符

- ✓由于二进制码0和1用于书写和阅读程序十分麻烦，常用3个或4个英文缩写来表示指令，这种缩写码叫做指令助记符
- ✓不同计算机指令系统的指令助记符的规定是不一样的
- ✓助记符到二进制操作码的转换——汇编程序（汇编器）

典型指令	指令助记符	二进制操作码
加法	ADD	001
减法	SUB	010
传送	MOV	011
跳转	JMP	100
转子	JSR	101
存储	STR	110
读数	LDA	111

主要内容

1. 指令系统概述
   - 1.1 指令系统的发展
   - 1.2 指令系统的性能要求
   - 1.3 编程语言与硬件的关系
2. 指令格式
   - 2.1 指令的一般格式
   - 2.2 指令字长
   - 2.3 指令助记符
3. 操作数与操作类型
   - 3.1 操作数类型
   - 3.2 数据在存储器中的存储方式
   - 3.3 操作类型
4. 寻址方式
   - 4.1 指令寻址
   - 4.2 数据寻址
5. CISC与RISC
   - 5.1 CISC技术
   - 5.2 RISC技术
   - 5.3 CISC与RISC的比较
6. 指令系统设计与举例
   - 6.1 典型指令
   - 6.2 典型指令系统
   - 6.3 指令系统设计及举例

3.1 操作数类型

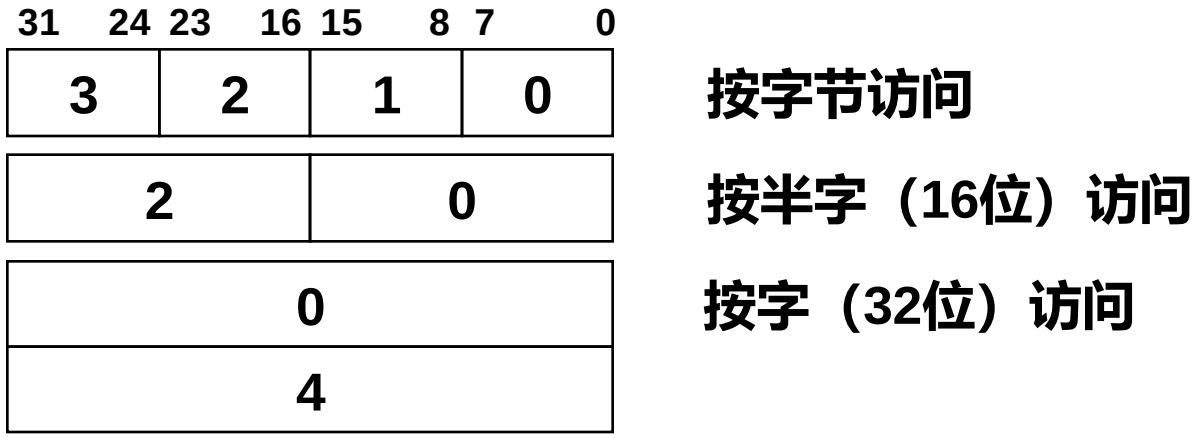
口计算机中常见的操作数类型

- ✓**地址：被看做一个无符号整数**
  - 很多情况下，对指令中的操作数的引用必须完成某种计算，才能确定其在主存中的有效地址
- ✓**数值：定点数、浮点数、十进制数等**
- ✓**字符：文本或字符串**
  - 普遍采用ASCII字符编码，以8位的字节来存储和传送字符
  - 其他的一些字符编码，如8位EBCDIC（扩展BCD交换码）
- ✓**逻辑数据**
  - 存储单元中的每一位“0”或“1”都代表真或假的布尔型值，这些0和1组合的数就被看做逻辑数
  - 能够对某个具体位进行逻辑运算

口关于数据存储的要求

- ✓存放在存储器或寄存器中
  - 寄存器位数反映机器字长
- ✓数据在存储器中一般是以字节（8位）为最小单位来存储
- ✓对数据的访问，可以按字节、半字、字或者双字进行

一个存储单元32位（存储字长）



多字节数据的存储引发的两个问题：

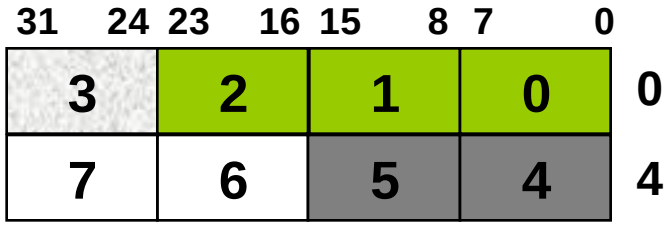
1. 边界对齐
2. 字节顺序

口 边界对齐问题

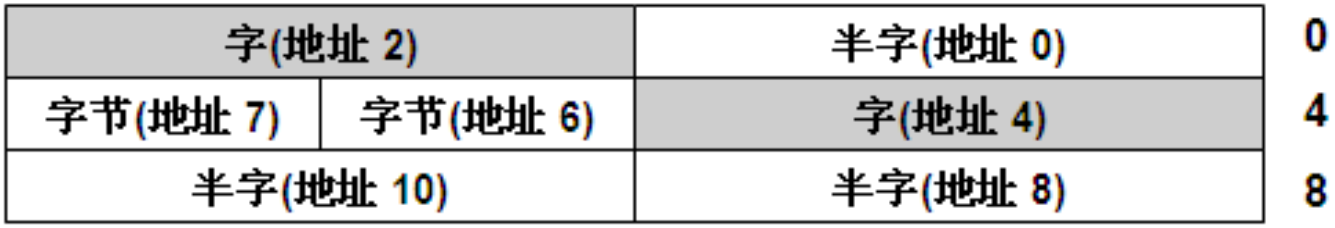
- ✓ 多字节的数据，在存储时需要“边界对齐”
- ✓ 因数据的访问方式而引发的存储限制

假设数据按半字访问，要存储两个数据，一个24位的A，一个16位的B

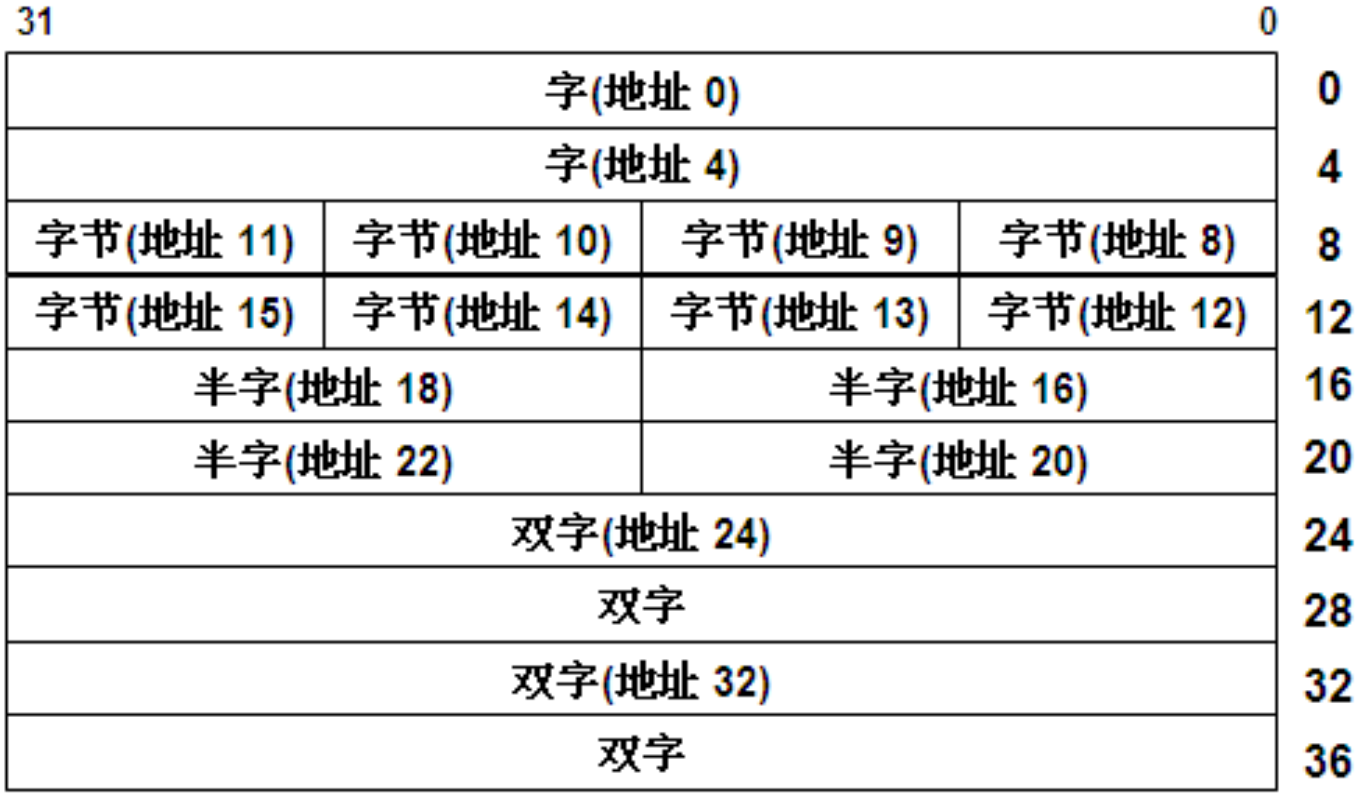
A存在0-2三个字节，因为边界对齐，3地址的字节放空，B存在4、5字节中



- ✓ 数据不做边界对齐带来的问题：读取一个数据时，可能需要两次访存操作才能取到完整的数据，降低了CPU的处理速度



3.2 数据的存放方式（3）



存储字长32位，对准边界的情况下，半字地址应是2的整数倍，字地址应该是4的整数倍，双字地址应该是8的整数倍

当所存储的数据不满足上述要求时，应填充一个或多个空白字节

3.2 数据的存放方式（4）

口字节顺序的问题

✓多字节数据的存储，涉及到

大尾端与小尾端 0X0123

字地址	31 24	23 16	15 8	7 0
0	0X0	0X1	0X2	0X3
4				

字地址

0

4

小尾端——低位在低地址

如下数据00000000 00000001 00000010 0000(

小尾端： 00000000 00000001 00000010 000

addr+3 addr+2 addr+1 ad

//低位在低地址

大尾端： 00000011 00000010 00000001 0000

addr+3 addr+2 addr+1 addr

//高位在低地址

```
#include <stdio.h>
int main()
{
    int tt = 1;
    char *c = (char*)(&tt);
    if(*c == 1)
    {
        printf(      endian\n");
    }
    else
    {
        printf(      endian\n");
    }
    return 0;
}
```

3.3 操作类型

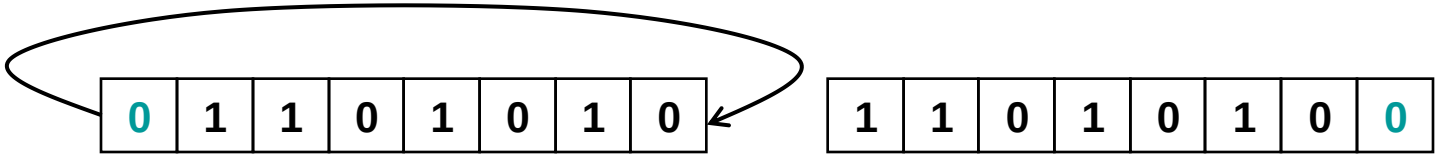
- 计算机指令系统的操作类型
  - ✓ 不同的机器，指令系统不尽相同，所支持的操作类型也就不同
  - ✓ 几乎所有机器都会具有一些通用的操作：**数据传送、算术逻辑运算、移位、跳转**等
- 数据传送操作
  - ✓ 寄存器与寄存器、寄存器与存储单元、存储单元与存储单元之间
  - ✓ 典型的操作：存储器读取**LOAD**和存入**STORE**；从源到目的的传送 **MOVE**；进栈**PUSH**、出栈**POP**
- 算术逻辑操作
  - ✓ 实现算术运算和逻辑运算
  - ✓ 一般机器都支持基本的二进制加减、比较等，有些支持浮点运算和十进制运算
  - ✓ 有些机器还支持位操作功能，如位测试、位清除、位求反等

3.3 操作类型（2）

口移位操作

- ✓分算术移位（有符号）、逻辑移位（无符号）、循环移位
- ✓算数移位常被用来代替简单的乘法和除法运算
  - 左移n位：乘以2^n；右移n位，除以2^n

循环移位

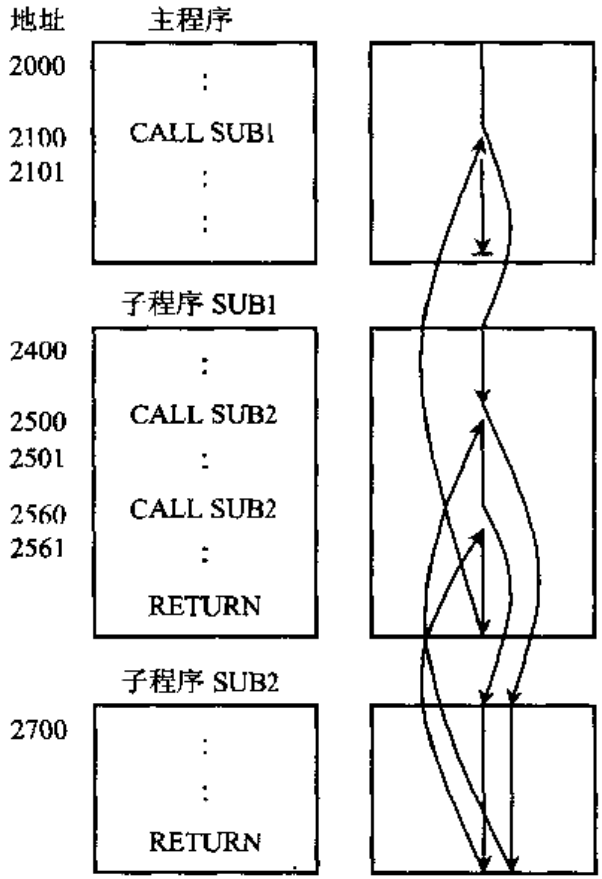


口转移跳转操作

- ✓一般情况下，计算机顺序执行指令，可以通过转移类指令改变
- ✓转移类指令(控制指令)：无条件转移、条件转移、过程调用与返回、陷阱等
  - 无条件转移：不受条件约束，直接把程序转移到指定的下一条指令位置继续执行
  - 条件转移：根据当前指令的执行结果来判断条件是否满足，从而决定是否进行转移

3.3 操作类型（3）

- 调用与返回：将一些特定的功能编写成独立的子程序，当程序中需要该功能时，通过调用指令进行调用，执行完后返回



(a) 主存空间分配　(b) 程序执行流程

- 子程序可以多处被调用
- 子程序调用可以嵌套
- 每条调用指令对应一条返回指令

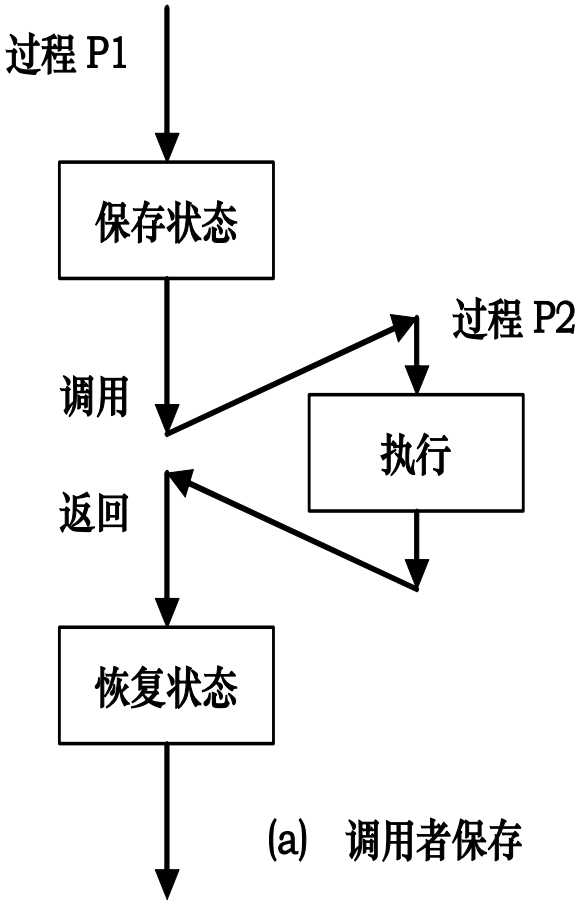
返回地址可能存放的地方：

- 寄存器：机器中设有专门的返回**地址寄存器**
- 子程序的**入口地址**中
- 栈顶：执行返回指令后，自动从**栈顶**取出返回地址

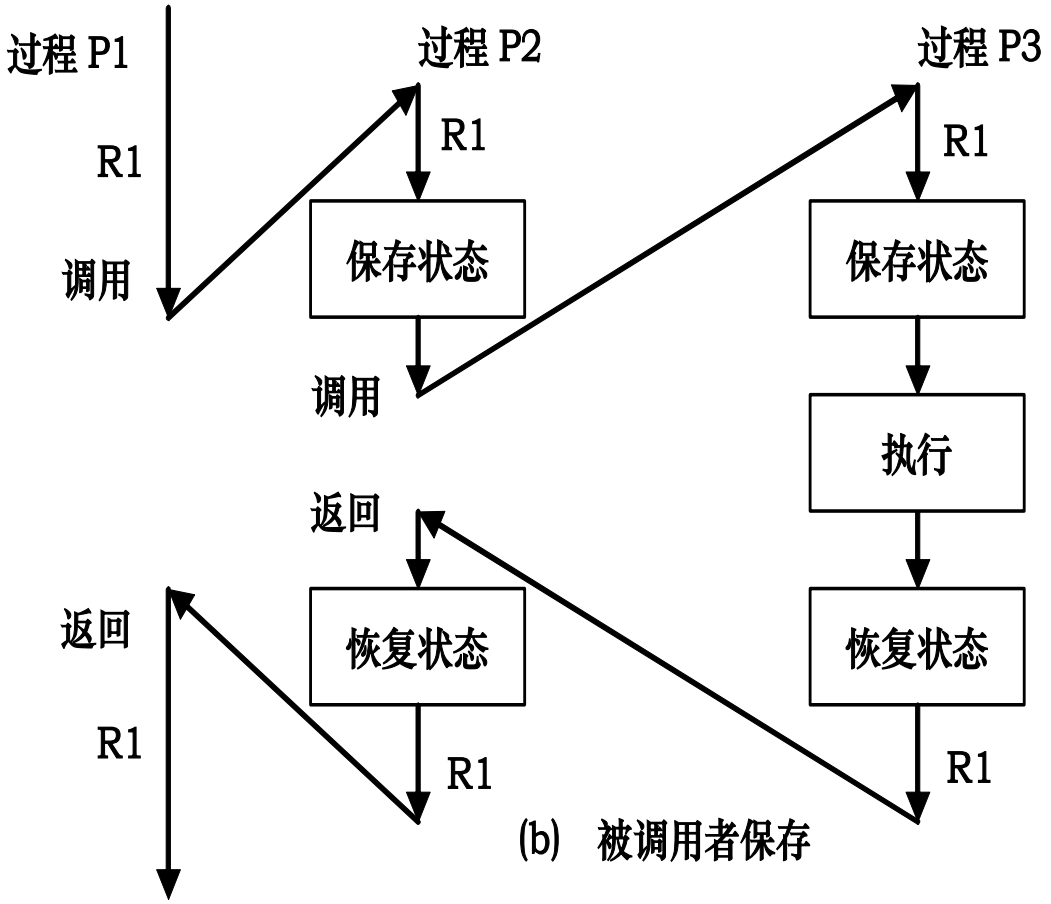
过程调用和返回的状态保存

- “调用者保存”（caller saving）方法：如果采用调用者保存策略，那么在一个调用者调用别的过程时，必须保存调用者所要保存的寄存器，以备调用结束返回后，能够再次访问调用者。
- “被调用者保存”（callee saving）方法：如果采用被调用者保存策略，那么被调用的过程必须保存它要用的寄存器，保证不会破坏过程调用者的程序执行环境，并在过程调用结束返回时，恢复这些寄存器的内容。

两种保存策略的比较

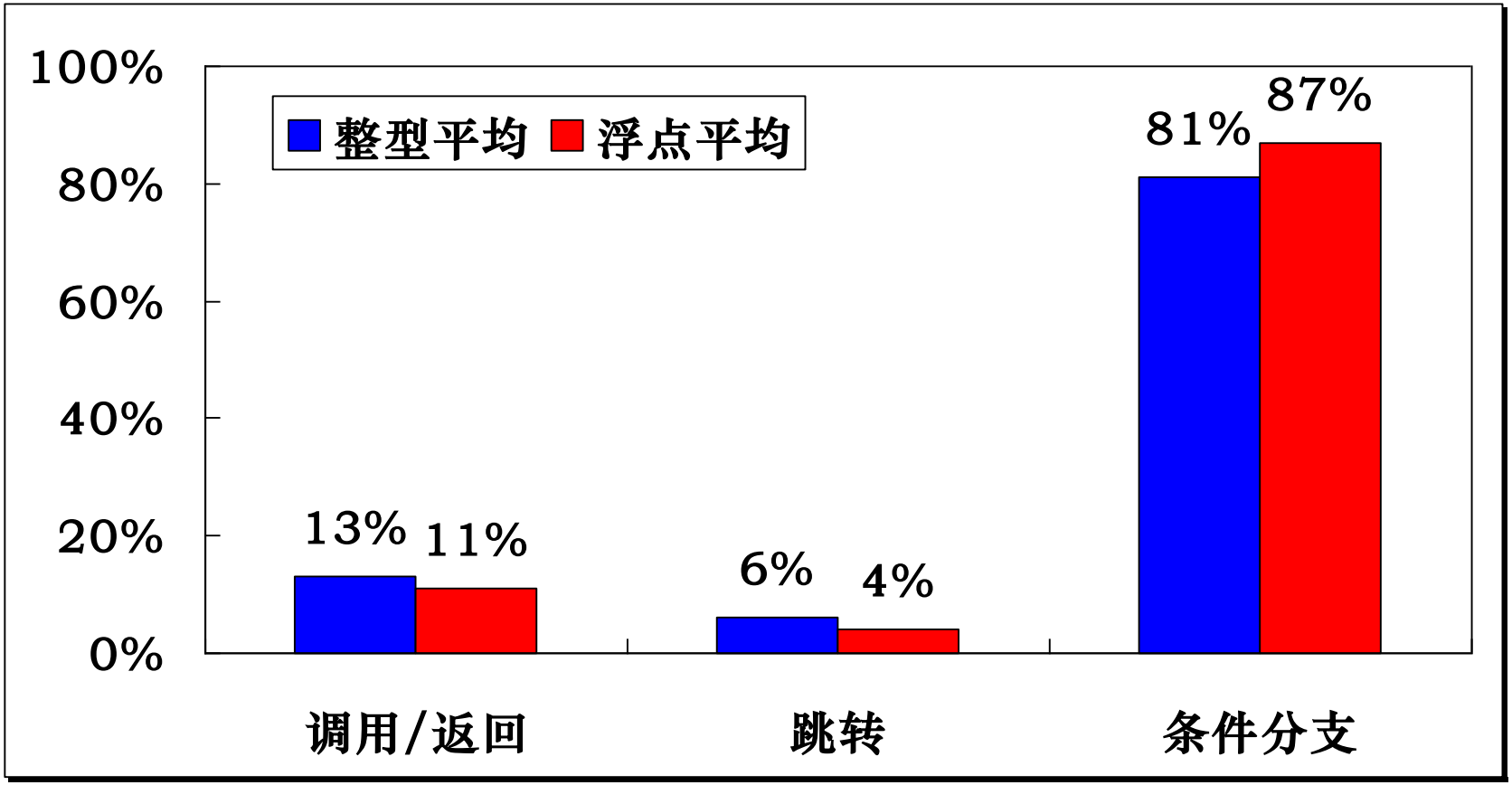


(a) 调用者保存



(b) 被调用者保存

控制指令的使用频率



3.3 操作类型（4）

- 陷阱：一种意外事故中断，如除0、运算溢出等。
  - ✓一般不提供给用户直接使用，而作为隐含指令由CPU在出现意外时自动产生并执行
  - ✓也有机器设置供用户使用的陷阱指令，利用它来完成系统调用和程序请求，如IBM PC (Intel 8086)的软中断INT TYPE
- 输入输出操作
  - ✓对于I/O独立编址的计算机，通过此类指令完成从外设中取数据和将数据输出到某个外设的操作
- 其他操作
  - ✓控制类：等待指令、停机指令、空指令、开/关中断指令等
  - ✓字符串传送、比较、查询及转换等
  - ✓向量指令

主要内容

1. 指令系统概述
   - 1.1 指令系统的发展
   - 1.2 指令系统的性能要求
   - 1.3 编程语言与硬件的关系
2. 指令格式
   - 2.1 指令的一般格式
   - 2.2 指令字长
   - 2.3 指令助记符
3. 操作数与操作类型
   - 3.1 操作数类型
   - 3.2 数据存储方式（边界对齐、存放顺序问题）
   - 3.3 操作类型（数据传送、算术逻辑运算、移位、跳转等）
4. 寻址方式
   - 4.1 指令寻址
   - 4.2 数据寻址
5. CISC与RISC
   - 5.1 CISC技术
   - 5.2 RISC技术
   - 5.3 CISC与RISC的比较
6. 指令系统设计与举例
   - 6.1 典型指令
   - 6.2 典型指令系统
   - 6.3 指令系统设计及举例

寻址方式

口地址

- ✓操作数或指令存放在某个**存储单元**时，该存储单元的编号，称为该操作数或指令在存储器中的地址

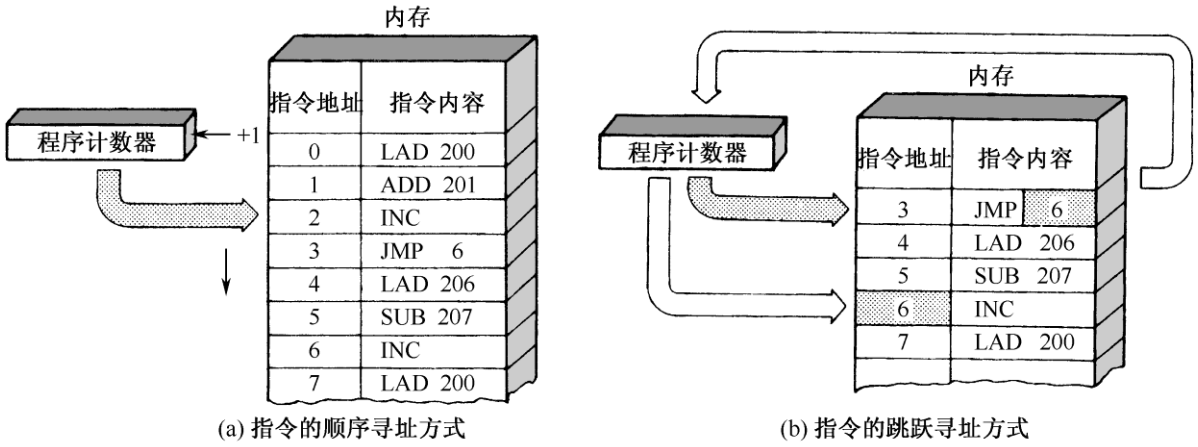
口寻址方式

- ✓确定本条指令中的**数据地址**以及下一条将要执行的**指令地址**的方法
- ✓与硬件结构紧密相关，直接影响指令格式和指令性能
- ✓分为**指令寻址**和**数据寻址**两类
  - 前者简单，后者复杂
  - 在冯·诺依曼型计算机中，**指令寻址**和**数据寻址**交替执行

4.1 指令寻址

口指令寻址

- ✓ 分为**顺序寻址**和**跳跃寻址**两种
- ✓ 顺序寻址
  - 通过程序计数器PC加1(+4...)，自动形成下一条指令的地址
- ✓ 跳跃寻址
  - 由当前指令（转移类指令）的地址码域给出下一条指令的地址
  - 跳跃后按新的指令地址再顺序寻址，直到又碰到转移类指令
  - 可以实现程序转移或构成循环程序



(a) 指令的顺序寻址方式　　(b) 指令的跳跃寻址方式

□数据寻址

✓形成数据的有效地址的方法，称为数据的寻址方式

- 形式地址A：指令的地址码字段所表示的地址，通常不代表操作数的**真实地址**
- 有效地址EA：相对于形式地址而言，是操作数的真实地址，由**寻址特征和形式地址**共同确定

✓寻址特征字段

操作码	寻址特征	形式地址A

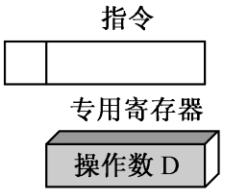
- 用于确定指令的操作数据应使用何种寻址方式
- 间接寻址位 + 变换寻址位 + 其他寻址位

数据寻址过程，就是把操作数的形式地址，通过寻址特征，变换为操作数的有效地址的过程

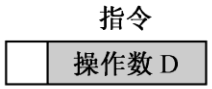
口基本的数据寻址方式分析

方式	算法	主要优点	主要缺点
隐含寻址	操作数在专用寄存器	无存储器访问	数据范围有限
立即寻址	操作数=A	无存储器访问	操作数幅值有限
直接寻址	EA=A	简单	地址范围有限
间接寻址	EA=(A)	大的地址范围	多重存储器访问
寄存器寻址	EA=R	无存储器访问	地址范围有限
寄存器间接寻址	EA=(R)	大的地址范围	额外存储器访问
偏移寻址	EA=A+(R)	灵活	复杂
段寻址（偏移）	EA=A+(R)	灵活	复杂
堆栈寻址	EA=栈顶	无存储器访问	应用有限

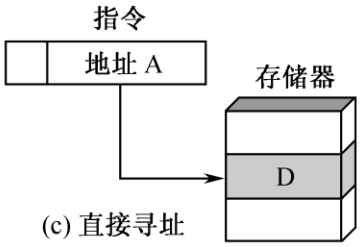
数据寻址方式示意图



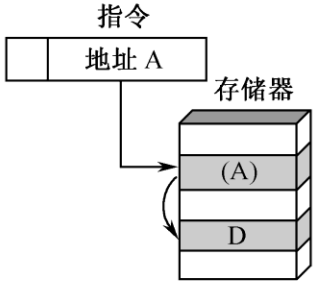
(a) 隐含寻址



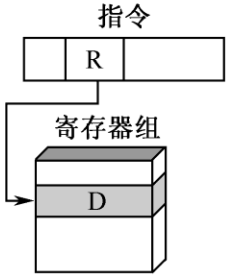
(b) 立即寻址



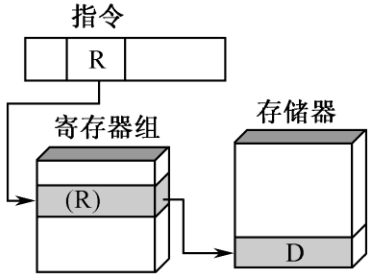
(c) 直接寻址



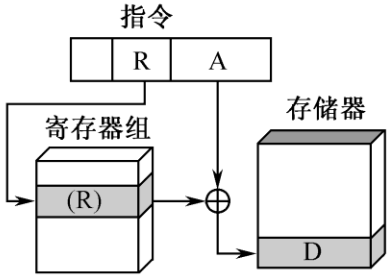
(d) 间接寻址



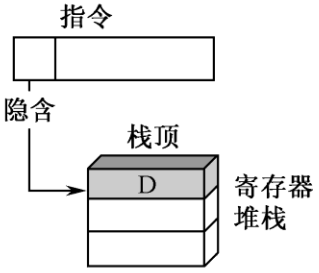
(e) 寄存器寻址



(f) 寄存器间接寻址



(g) 偏移寻址

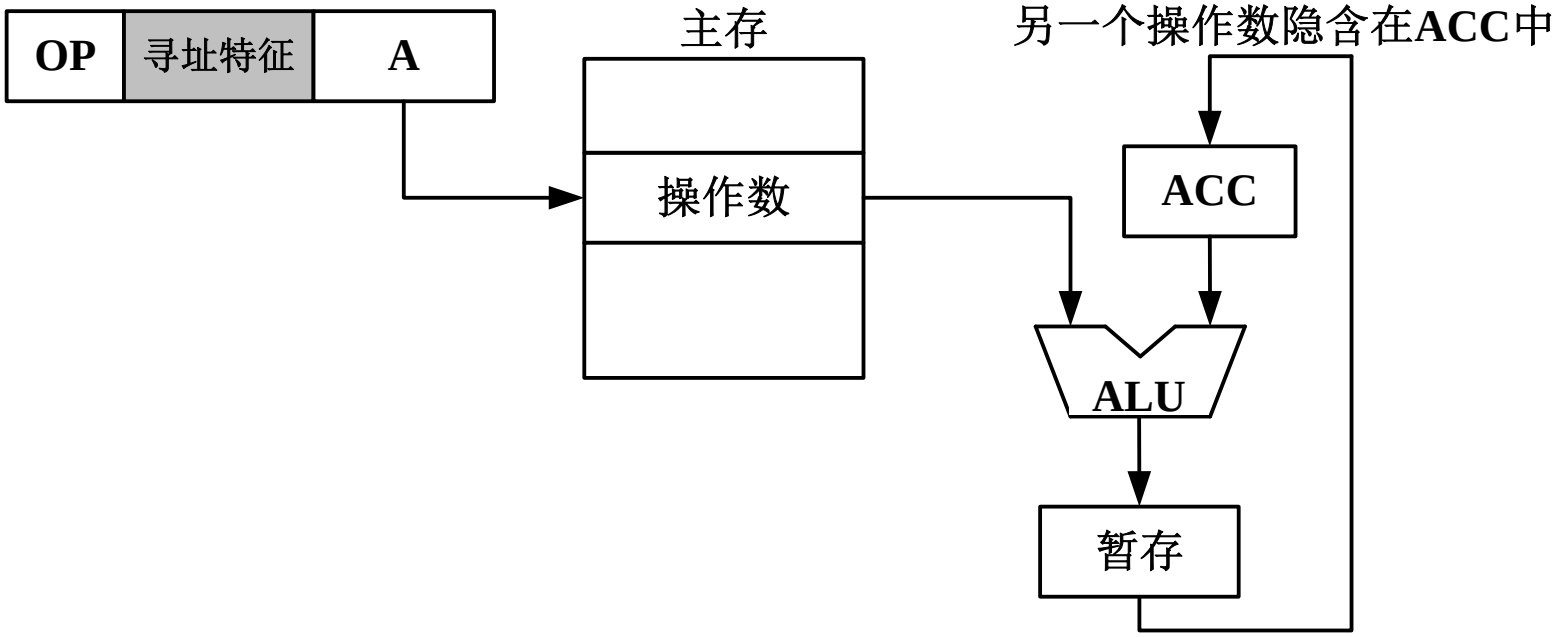


(h) 堆栈寻址

4.2 数据寻址——隐含寻址

□隐含寻址

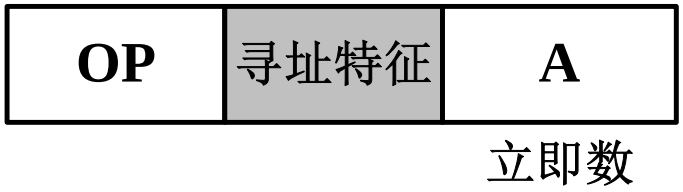
- ✓指令中不明显给出操作数的地址，而是隐含在特定的寄存器中
  - 如单地址指令，第二操作数地址隐含在累加寄存器ACC中
- ✓有利于缩短指令字长



4.2 数据寻址——立即寻址

口立即寻址

- ✓指令的地址字段指出的不是操作数地址，而是操作数本身
  - 指令中的操作数又称为立即数，一般采用**补码形式**存放
- ✓取出指令就可同时获得操作数，不必访问内存
- ✓A的位数限制了立即数的取值范围



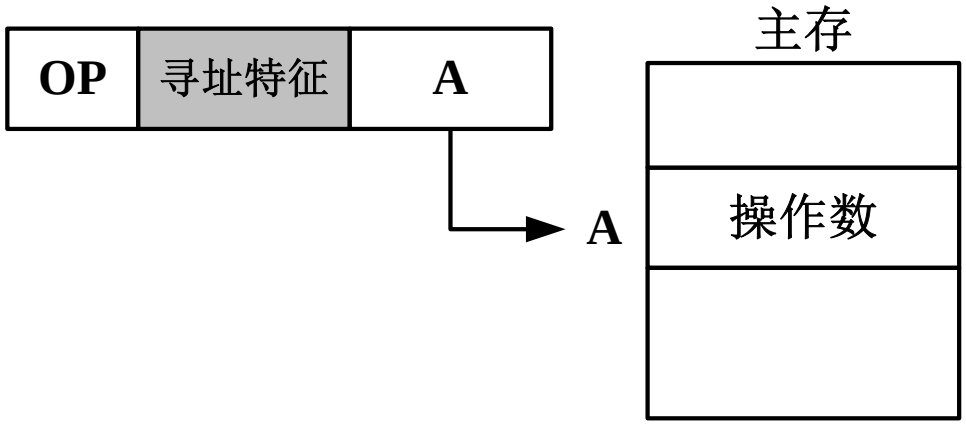
4.2 数据寻址——直接寻址

口直接寻址

- ✓指令字中的形式地址A就是真实地址EA

$$EA = A$$

- ✓优点：寻找操作数比较简单，无需专门计算操作数地址，访存一次即取得操作数
- ✓缺点：A的位数限制了操作数的寻址范围

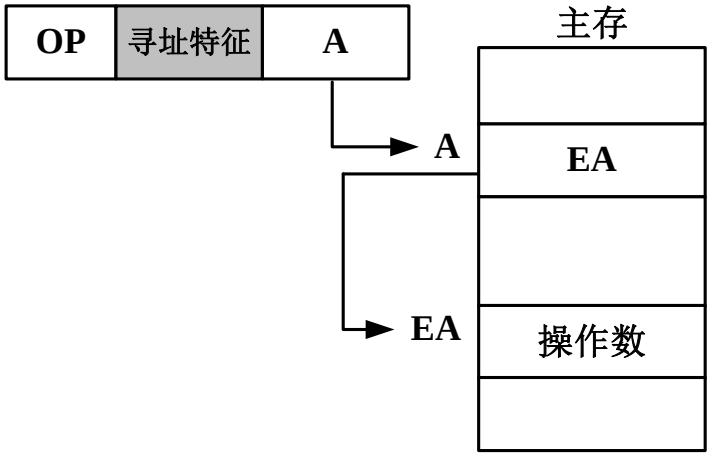


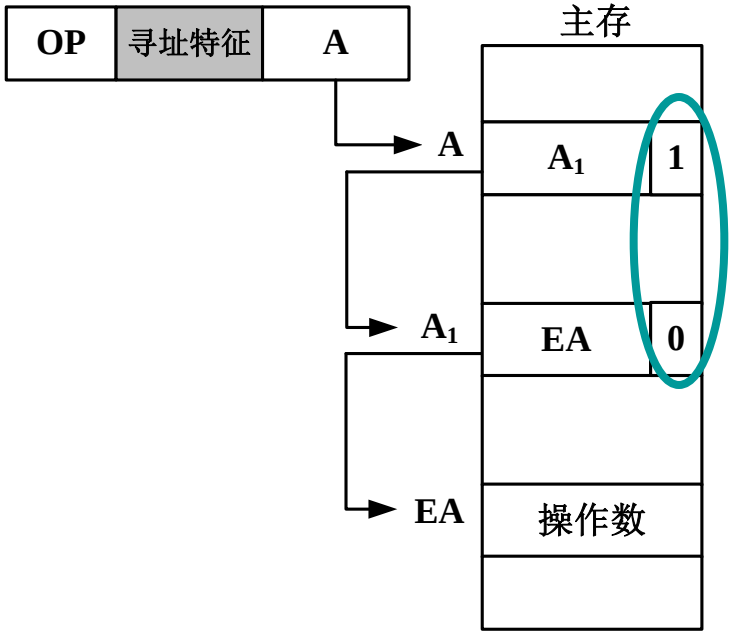
口间接寻址

- ✓形式地址A指出操作数有效地址EA所在的存储单元，即有效地址由形式地址间接提供

$$EA = (A)$$

- ✓优点：1）扩大了操作数的寻址范围；

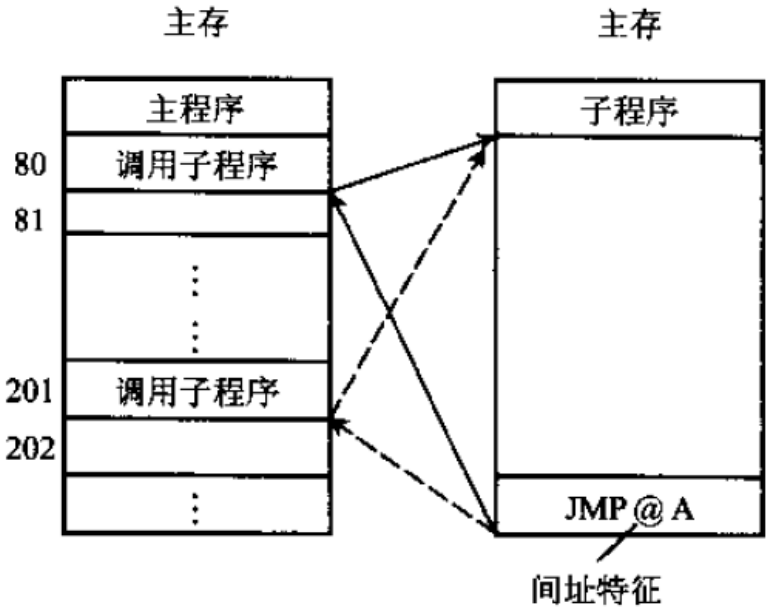




4.2 数据寻址——间接寻址（2）

✓优点：2）便于编程

- 调用子程序前，将返回地址存入子程序最末条指令的形式地址的存储单元，便可准确返回原程序



第一次调用前，将 81 → A

第二次调用前，将 202 → A

✓缺点：需要两次或多次访存，指令执行时间变长

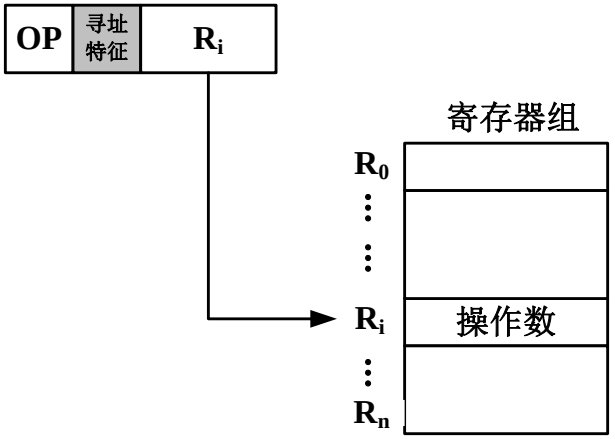
4.2 数据寻址
—寄存器寻址与寄存器间接寻址

□寄存器寻址

- ✓ 操作数不在内存中，而放在CPU的通用寄存器里；形式地址A表示的是寄存器的编号R，即

  EA = R

- ✓ 无须访存，减少了指令执行时间；
- ✓ 因寄存器编号较短，故可以压缩指令字，节省存储空间

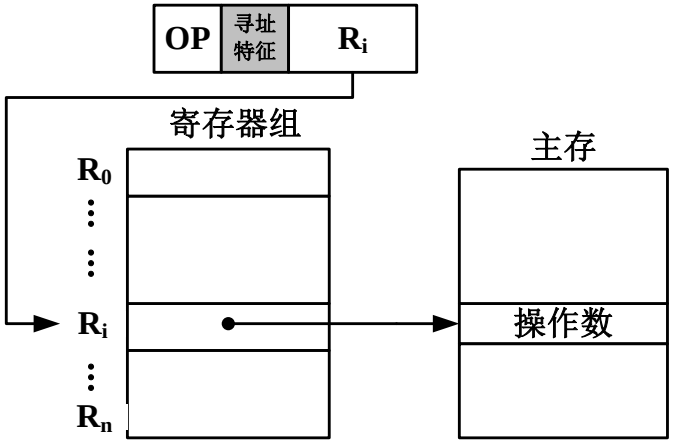


□寄存器间接寻址

- ✓ 形式地址A所表示的寄存器中，存放的不是实际操作数，而是操作数在内存中的地址，即

  EA = (R)

- ✓ 需访存操作，不过相比于间接寻址，减少了一次访存



偏移寻址

✓ 直接寻址 + 寄存器间接寻址

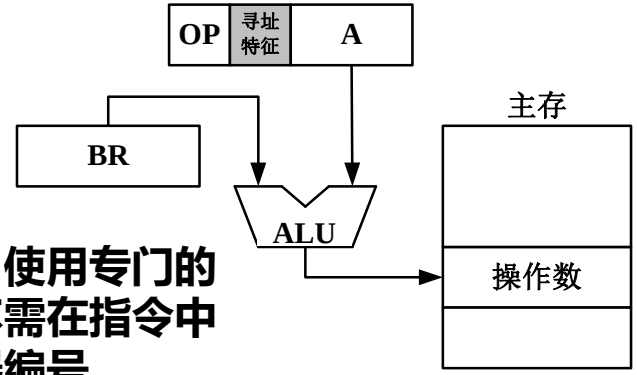
$$EA = A + (R)$$

一般有两个地址字段：

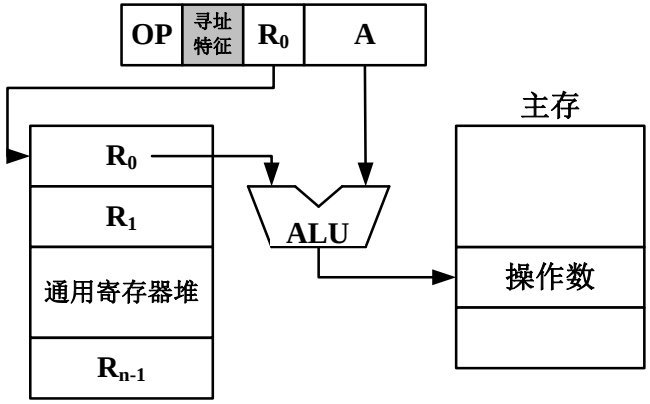
A，内存形式地址，直接使用；

R，某寄存器编号，间接使用

✓ 根据使用的寄存器是专用或通用寄存器，可分为隐式或显式两种



隐式偏移，使用专门的寄存器，不需在指令中指明寄存器编号



✓ 常用的三种形式：相对寻址、基址寻址、变址寻址

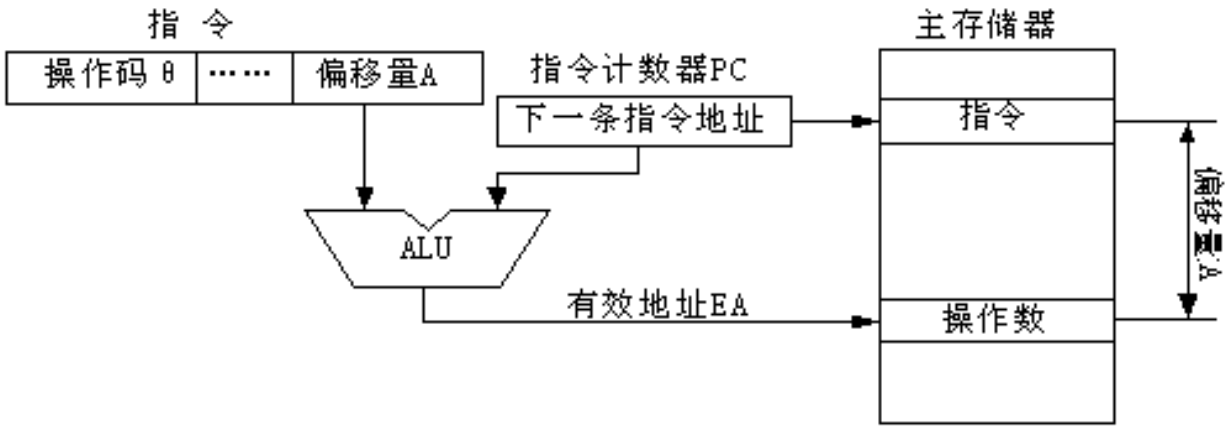
4.2 偏移寻址——相对寻址

□相对寻址

✓使用程序计数器PC提供基准地址，指令字中的形式地址A给出相对位移量，即

$$EA = A + (PC)$$

- A称为偏移量，通常用补码表示
- 有效地址是相对当前指令地址的一定范围内的偏移——基于**程序的局部性原理**



口程序的局部性原理

- ✓程序的执行过程，总是趋向于使用最近使用过的**数据和指令**，也就是说程序执行时所使用的信息和访问的存储器地址分布不是随机的，而是相对地集中
- ✓这种集中包括**时间和空间**两个方面：
  - **程序的时间局部性**——指如果程序中的某条指令一旦执行，则不久之后该指令可能再次被执行；如果某数据被访问，则不久之后该数据可能再次被访问（循环）
  - **程序的空间局部性**——指一旦程序访问了某个存储单元，则不久之后，其附近的存储单元也将被访问（数组）

4.2 偏移寻址—相对寻址（2）

✓相对寻址方式的用处

- 代码模块可采用浮动地址，程序在内存中可以任意放置
- 编程只需确定程序内部操作数与指令之间的**相对距离**，而无需确定操作数在主存储器中的**绝对地址**，这样，程序可以安排在主存的任意位置而不会影响其正确性
- 常用于转移类指令，转移地址随PC值变化

```
…
M      ADD  X, D
M+1    INX
M+2    CPX  #N
M+3    BNE  M
```

指令中M表示的地址，随整段程序在内存中的存放位置的不同而需要进行修改

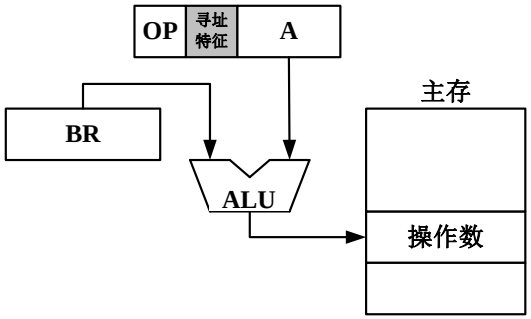
采用相对寻址，将“BNE M”改为“BNE * -3”（假设*为相对寻址特征），就能够避免这种修改

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

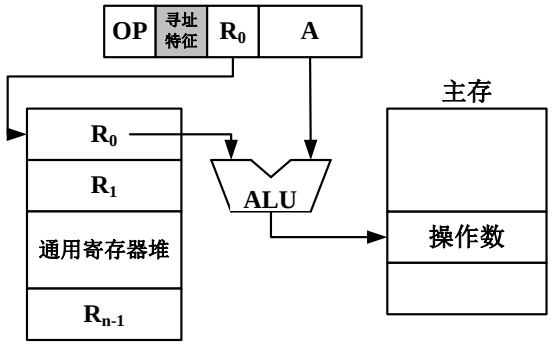
4.2 偏移寻址——基址寻址与变址寻址

□基址寻址 与 变址寻址

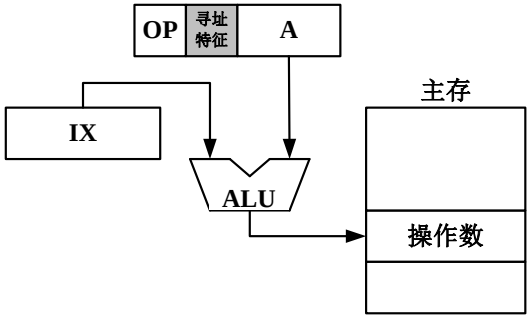
- ✓两者有效地址的形成过程极为相似，即 EA = A + (R)
- ✓使用哪个寄存器用作基址/变址寄存器，需在硬件设计时明确指定



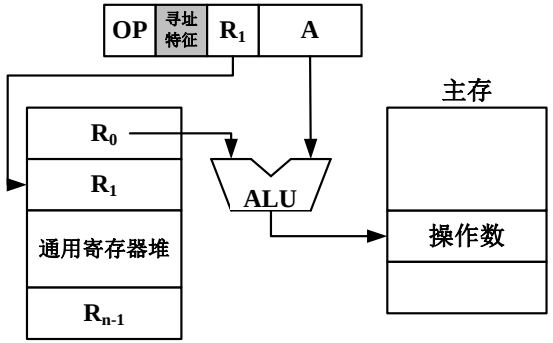
a) 专用基址寄存器BR



b) 通用寄存器用作基址寄存器



a) 专用变址寄存器BR



b) 通用寄存器用作变址寄存器

4.2 偏移寻址——基址寻址与变址寻址（2）

寻址本质

- ✓ 基址寻址——寄存器中含有一个主存地址，指令中的形式地址A表示相对于该地址的偏移量
  - 基址寄存器的内容通常由操作系统或管理程序确定，在程序执行过程中不可变，可变的是形式地址A
- ✓ 变址寻址——指令中的形式地址A表示一个主存地址，寄存器中含有相对于该地址的偏移量
  - 变址寄存器的内容由用户设定，在程序执行过程中可变，但形式地址A的内容是不可变的

应用场合

- ✓ 基址寻址的用处
  - 扩大操作数的寻址范围——基址寄存器位数可设置得很长
  - 用于为程序或数据分配存储空间，实现存储透明性
  - 实现段寻址
- ✓ 变址寻址的用处
  - 常用于需要频繁修改操作数地址的处理，如数组运算、字符串操作以及循环重复等

关于基址寻址的进一步说明

- 好处1：基址寻址可以扩大操作数的寻址范围
  - ✓ 主存容量较大时，采用直接寻址无法对主存所有单元进行访问
  - ✓ 而采用基址寻址，由于**基址寄存器的位数可以大于形式地址位数**，可以实现对主存空间的更大范围的访问
- 好处2：存储透明性
  - ✓ 用户不必考虑自己的程序在主存中的位置，由**操作系统**或管理程序根据主存情况，赋予**基址寄存器一个初始基地址**
  - ✓ 用户在程序执行过程中不可修改基址寄存器的内容
- 用途：段寻址方式
  - ✓ 将主存空间分为若干段，每段的首地址存于基址寄存器，段内偏移量由指令字中的形式地址A指出

关于变址寻址的进一步说明

例 求具有N个数的一个数组D的平均值

直接寻址求 N 个数的平均值程序

程　序	说　明
LDA　D	[D] →ACC
ADD　D+1	[ACC]+[D+1] →ACC
ADD　D+2	[ACC]+[D+2] →ACC
⋮	⋮
ADD　D+(N−1)	[ACC]+[D+(N−1)] →ACC
DIV　#N	[ACC]÷N →ACC
STA　ANS	[ACC] →ANS 单元（ANS 为内存某单元地址）

程序所用的指令数为N+2

变址寻址求 N 个数的平均值程序

程　序	说　明
LDA　#0	0 →ACC
LDX　#0	0 →X（X 为变址寄存器）
M　ADD　X,D	[ACC]+[D+(X)] →ACC（D 为形式地址，X 为变址寄存器）
INX	[X]+1 →X
CPX　#N	[X]－N，并建立 Z 的状态，结果为“0” Z=1；结果非“0” Z=0
BNE　M	当 Z=1 时，按顺序执行；当 Z=0 时，转至 M
DIV　#N	[ACC]÷N →ACC
STA　ANS	[ACC] →ANS（ANS 为内存某单元地址）

仅用8条指令，且不因N的增大而增加指令

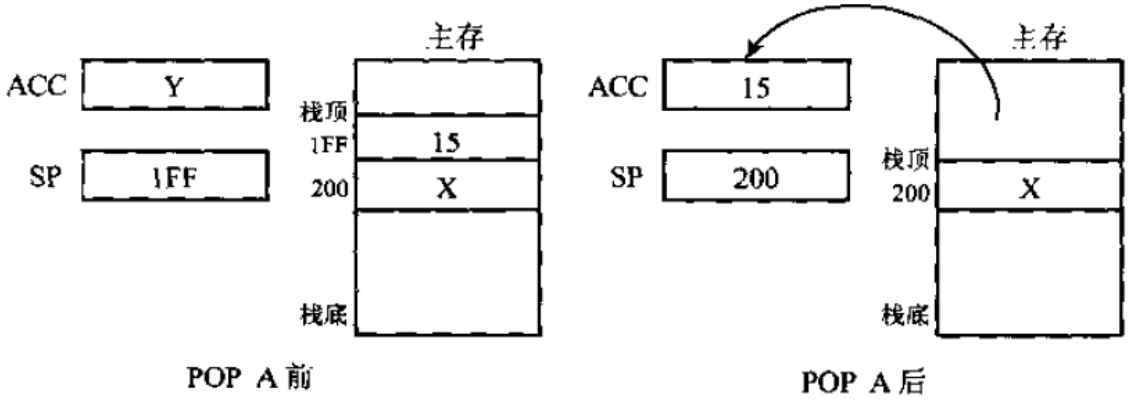
基址寻址 V.S.变址寻址

- 基址寻址(Base or Displacement addressing)
  - ✓以基址寄存器（BR）为基准进行寻址
  - ✓EA = 形式地址 + BR
  - ✓基址寄存器：专用、通用
    - 显式（通用寄存器）、隐式（专用寄存器）
- 变址寻址（index)
  - ✓以变址寄存器（ IX ）的值为基准寻址
  - ✓EA = 形式地址 + IX
  - ✓变址寄存器：专用、通用
    - 显式（通用寄存器）、隐式（专用寄存器）
- 基址寻址 vs. 变址寻址
  - ✓基址：BR由OS赋值，不变；形式地址可变
  - ✓变址：IX由程序员赋值，可变；形式地址不变
  - ✓用途不同：基址—段寻址，变址—数组、字符串、循环

4.2 数据寻址——堆栈寻址

□ 堆栈寻址

- ✓ 要求计算机系统中设有堆栈才能够实现
- ✓ 两种形式：硬堆栈，寄存器组；软堆栈，主存的一部分空间
- ✓ 先进后出，数据存取只在栈顶地址进行
  - 使用一个隐式或显式的堆栈指针SP——堆栈寄存器
  - 数据进栈或出栈，相应地修改SP的值
- ✓ 可视为是一种隐含寻址，操作数地址隐含在SP中；也可视为寄存器间接寻址，SP为寄存器，存放操作数有效地址

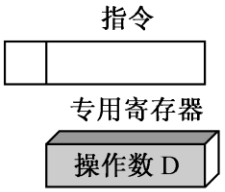


数据寻址小结

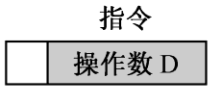
口基本的数据寻址方式分析

方式	算法	主要优点	主要缺点
隐含寻址	操作数在专用寄存器	无存储器访问	数据范围有限
立即寻址	操作数=A	无存储器访问	操作数幅值有限
直接寻址	EA=A	简单	地址范围有限
间接寻址	EA=(A)	大的地址范围	多重存储器访问
寄存器寻址	EA=R	无存储器访问	地址范围有限
寄存器间接寻址	EA=(R)	大的地址范围	额外存储器访问
偏移寻址	EA=A+(R)	灵活	复杂
段寻址（偏移）	EA=A+(R)	灵活	复杂
堆栈寻址	EA=栈顶	无存储器访问	应用有限

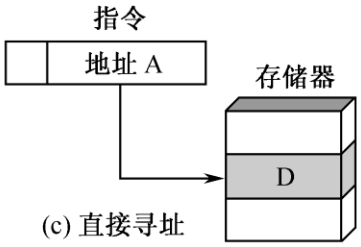
数据寻址方式示意图



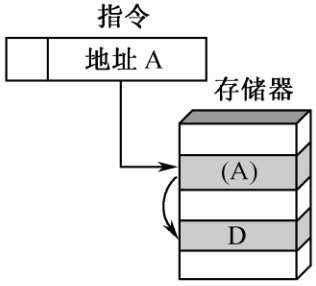
(a) 隐含寻址



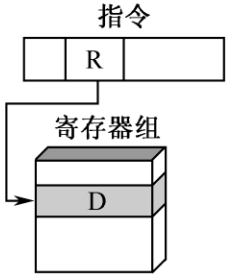
(b) 立即寻址



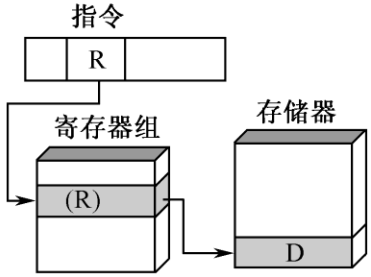
(c) 直接寻址



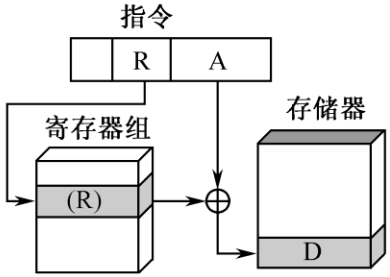
(d) 间接寻址



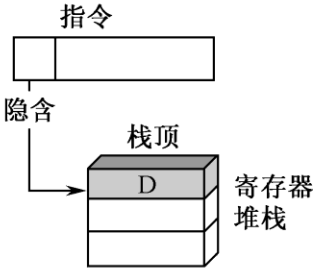
(e) 寄存器寻址



(f) 寄存器间接寻址



(g) 偏移寻址



(h) 堆栈寻址

基址寻址 V.S.变址寻址

- 基址寻址(Base or Displacement addressing)
  - ✓以基址寄存器（BR）为基准进行寻址
  - ✓EA = 形式地址 + BR
  - ✓基址寄存器：专用、通用
    - 显式（通用寄存器）、隐式（专用寄存器）
- 变址寻址（index)
  - ✓以变址寄存器（ IX ）的值为基准寻址
  - ✓EA = 形式地址 + IX
  - ✓变址寄存器：专用、通用
    - 显式（通用寄存器）、隐式（专用寄存器）
- 基址寻址 vs. 变址寻址
  - ✓基址：BR由OS赋值，不变；形式地址可变
  - ✓变址：IX由程序员赋值，可变；形式地址不变
  - ✓用途不同：基址—段寻址，变址—数组、字符串、循环

常用的一些操作数寻址示例

- 寄存器寻址
  - ✓指令实例：Add R4 , R3
  - ✓含义：Regs[R4]←Regs[R4] + Regs[R3]
- 立即寻址
  - ✓指令实例：Add R4 , #3
  - ✓含义：Regs[R4]←Regs[R4] + 3

常用的一些操作数寻址方式

- 直接寻址
  - ✓指令实例：Add R1，(1001)
  - ✓含义：
    - Regs[R1]←Regs[R1] + Mem[1001]
- 寄存器间接寻址
  - ✓指令实例：Add R4，(R1)
  - ✓含义：
    - Regs[R4]←Regs[R4] + Mem[Regs[R1]]

常用的一些操作数寻址方式

- 存储器间接寻址
  - ✓指令实例：Add R1，@(R3)
  - ✓含义：
    - Regs[R1]←Regs[R1] + Mem[Mem[Regs[R3]]]
- 偏移寻址
  - ✓指令实例：Add R4，100(R1)
  - ✓含义：
    - Regs[R4]←Regs[R4] + Mem[100+Regs[R1]]

常用的一些操作数寻址方式

- 自增寻址
  - ✓指令实例：Add R1，(R2)+
  - ✓含义：
    - Regs[R1]←Regs[R1] + Mem[Regs[R2]]
    - Regs[R2]←Regs[R2] + d
- 自减寻址
  - ✓指令实例：Add R1, -(R2)
  - ✓含义：
    - Regs[R2]←Regs[R2] - d
    - Regs[R1]←Regs[R1]+Mem[Regs[R2]]

常用的一些操作数寻址方式

- 索引寻址
  - ✓指令实例：Add R3，(R1 + R2)
  - ✓含义：
    - Regs[R3]←Regs[R3] + Mem[Regs[R1]+Regs[R2]]
- 缩放寻址
  - ✓指令实例：Add R1，100(R2)[R3]
  - ✓含义：
    - Regs[R1]←Regs[R1] + Mem[100 + Regs[R2] + Regs[R3]*d]

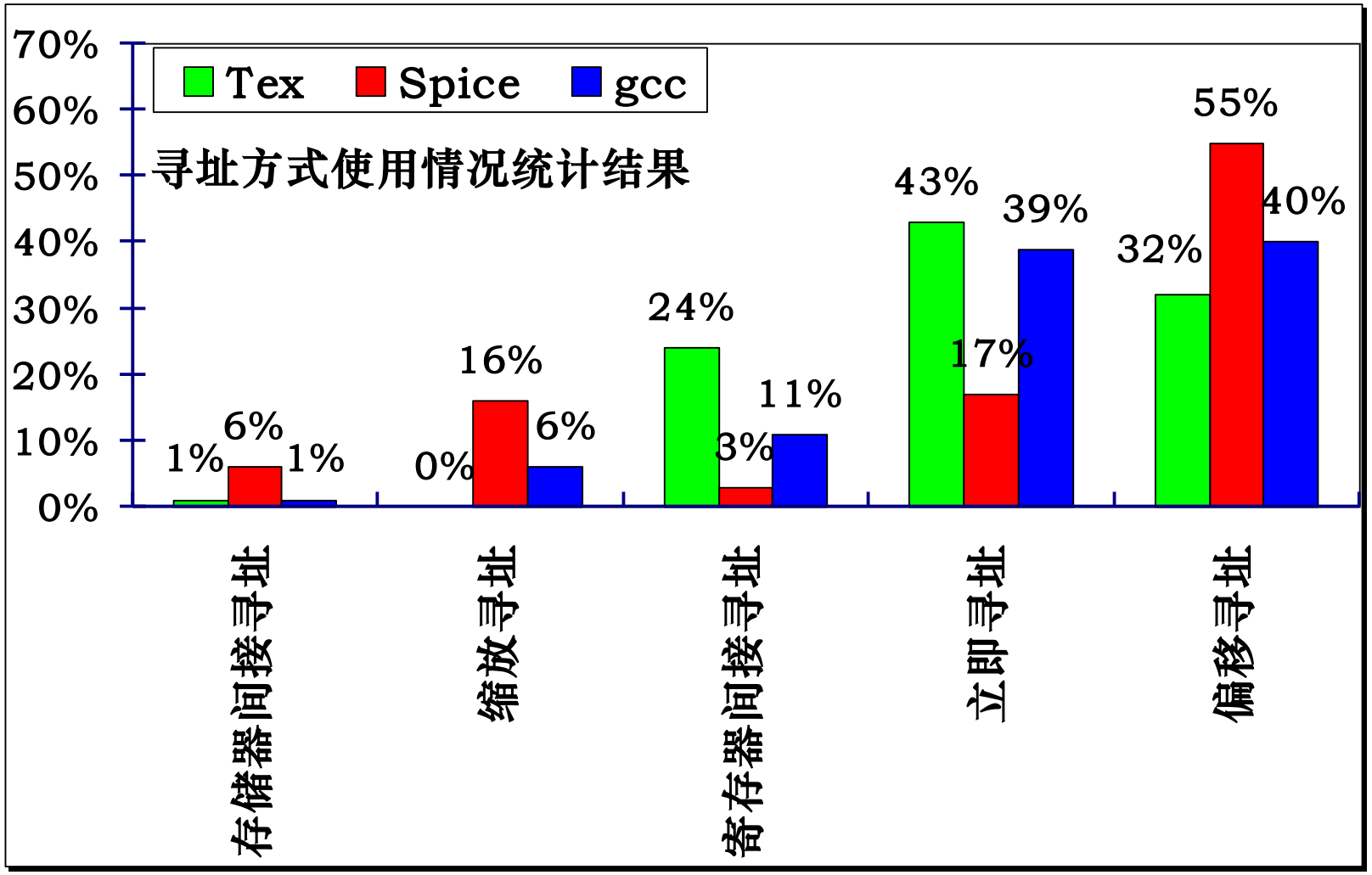
寻址方式总结

- 硬件：当今计算机种类繁多，指令系统各不相同，**各类机器的寻址方式有着各自的特点，机器指令系统设计的重点**

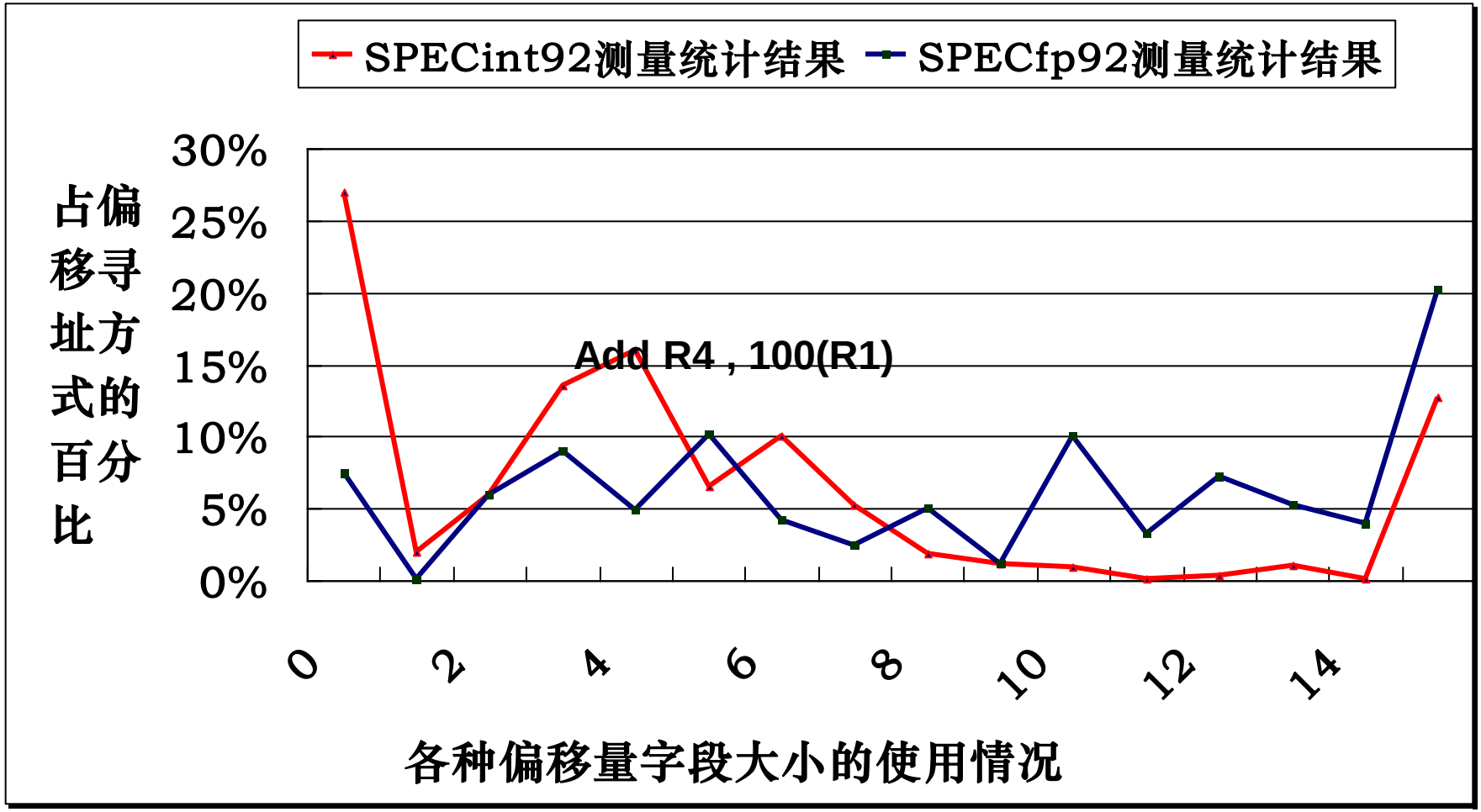
- 软件：从高级语言编程角度考虑，寻址方式对用户无关紧要；但若采用**汇编语言编程**的设计，则了解和掌握机器的寻址方式，是成功完成工作必不可少的

- 系统：透彻了解**机器指令的寻址方式，能使得自身进一步加深对机器内信息流程和整机工作概念的理解**，对提高工作效率和系统性能有很大的帮助

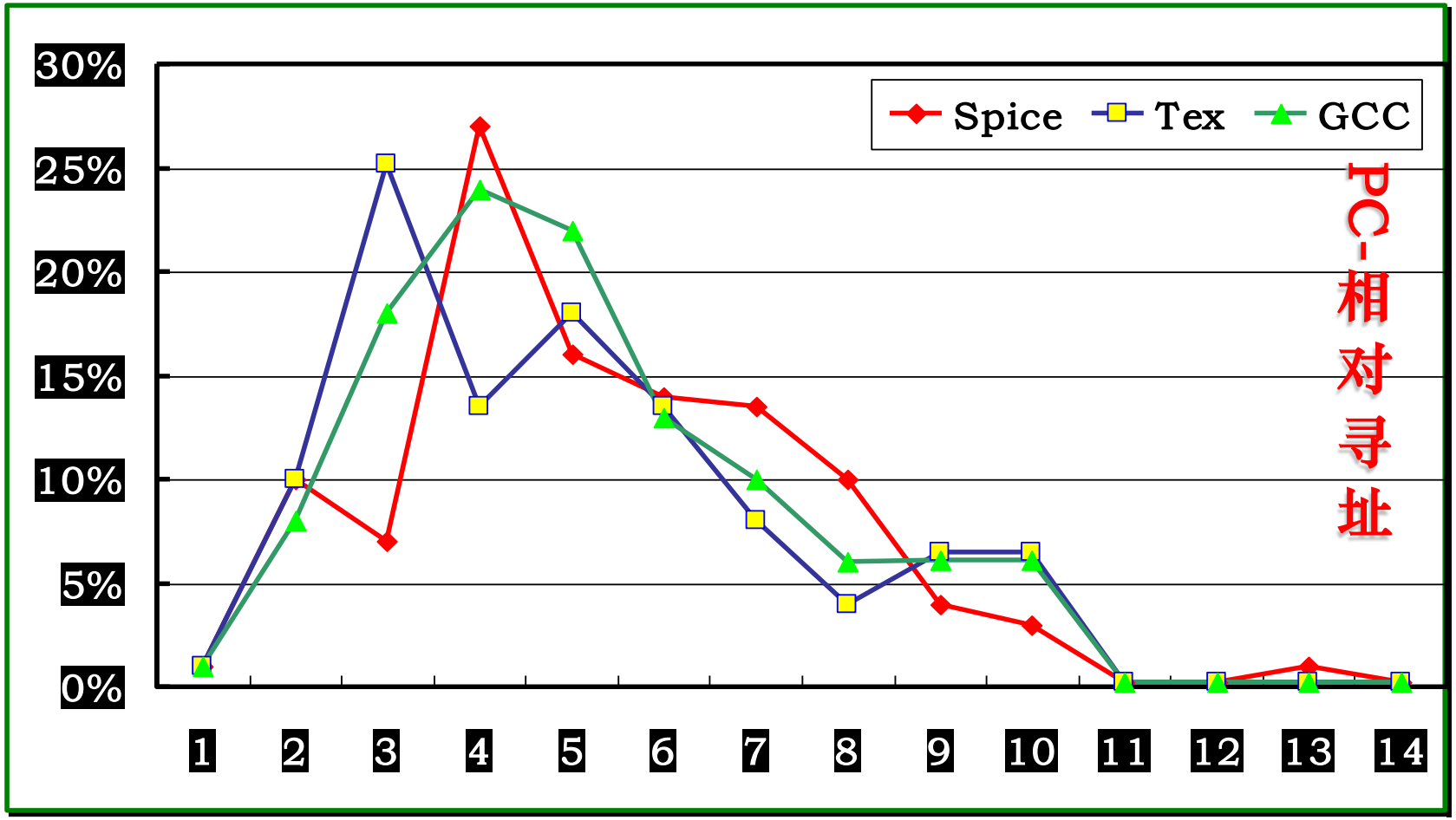
常用的一些操作数寻址方式



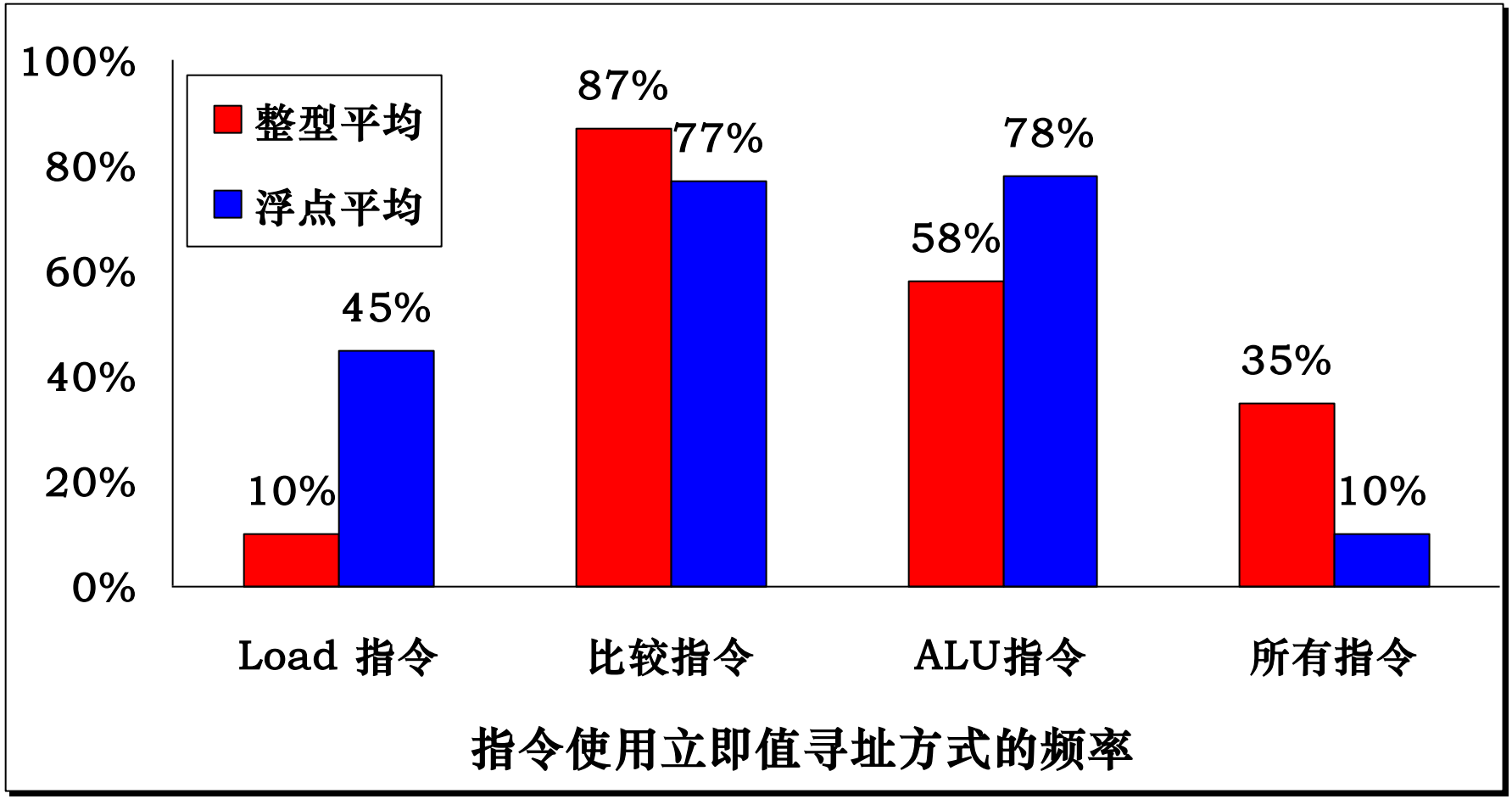
偏移寻址


SPECint92测量统计结果
SPECfp92测量统计结果
占偏移寻址方式的百分比
30%
25%
20%
15%
10%
5%
0%
Add R4 , 100(R1)
0
2
4
6
8
10
12
14
各种偏移量字段大小的使用情况

偏移寻址-相对寻址
分支目标地址的表示

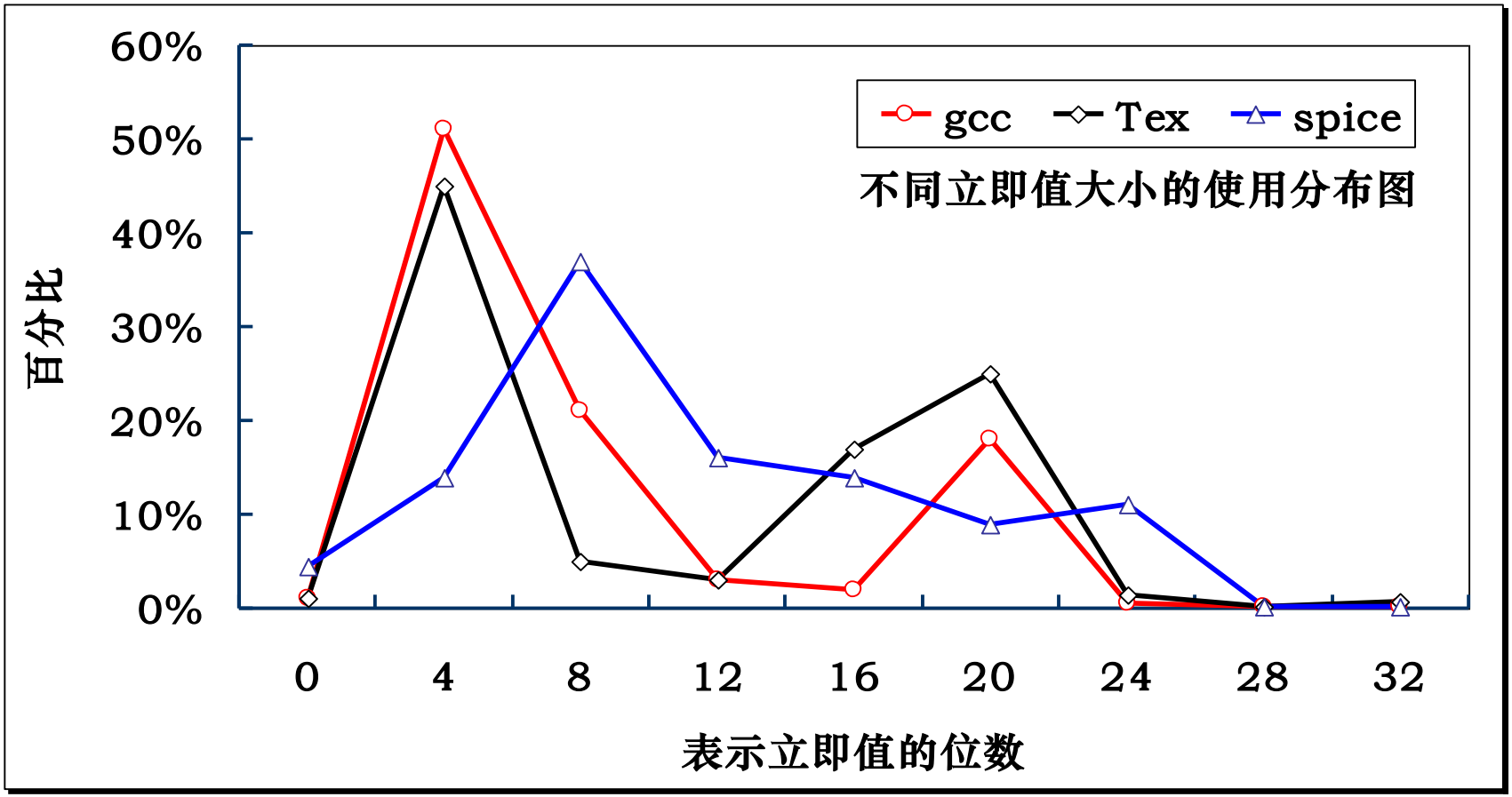


立即寻址



指令使用立即值寻址方式的频率

立即寻址


60%
50%
40%
30%
20%
10%
0%
百分比
gcc
Tex
spice
不同立即值大小的使用分布图
0
4
8
12
16
20
24
28
32
表示立即值的位数

主要内容

1. 指令系统概述
   - 1.1 指令系统的发展
   - 1.2 指令系统的性能要求
   - 1.3 编程语言与硬件的关系
2. 指令格式
   - 2.1 指令的一般格式
   - 2.2 指令字长
   - 2.3 指令助记符
3. 操作数与操作类型
   - 3.1 操作数类型
   - 3.2 数据在存储器中的存储方式
   - 3.3 操作类型
4. 寻址方式
   - 4.1 指令寻址
   - 4.2 数据寻址
5. CISC与RISC
   - 5.1 CISC技术
   - 5.2 RISC技术
   - 5.3 CISC与RISC的比较
6. 指令系统设计与举例
   - 5.1 典型指令
   - 5.2 典型指令系统
   - 5.3 指令系统设计及举例

指令集结构功能设计方向

- 一个方向是强化指令功能，实现软件功能向硬件功能转移，基于这种指令集结构而设计实现的计算机系统称为**复杂指令集计算机**（CISC）。
- 另一个方向是八十年代发展起来的**精简指令集计算机**（RISC），其目的是尽可能地降低指令集结构的复杂性，以达到简化实现，提高性能的目的。

5.1 CISC技术

- Complex Instruction Set Computer——复杂指令集计算机
  - ✓形成原因
    - 早期计算机部件昂贵，主频低，运算慢；
    - 为提高速度，将越来越多的复杂指令加入到指令系统中
  - ✓特点
    - 指令系统庞大，指令功能丰富
    - 指令格式和寻址方式较多，多数指令需多个指令周期完成
    - 各种指令都可以访问存储器，少量的专用寄存器
    - 通过复杂化硬件来简化软件实现
  - ✓相关技术
    - 操作码扩展
    - 各种寻址方式

CISC指令集功能设计

- 面向目标程序增强指令功能
  - ✓ 提高运算型指令功能；
  - ✓ 提高传送指令功能；
  - ✓ 增加程序控制指令功能。
- 面向操作系统和编译程序改进指令系统
  - ✓ 主要表现在对中断处理、进程管理、存储管理和保护、系统工作状态的建立与切换、进程同步和互斥等的支持。
  - ✓ 增加对编译系统支持的指令功能

RISC指令集功能设计

- CISC结构存在着如下缺点：
  - ✓ 在CISC结构的指令系统中，各种指令的使用频率相差悬殊。据统计，有20%的指令使用频率最大，占运行时间的80%。也就是说，有80%的指令在20%的运行时间内才会用到。
  - ✓ CISC结构指令系统的复杂性带来了计算机体系结构的复杂性，这不仅增加了研制时间和成本，而且还容易造成设计错误。CISC结构指令系统的复杂性给VLSI设计增加很大负担，不利于单片集成。许多指令需要很复杂的操作，因而运行速度慢。
  - ✓ 在CISC结构的指令系统中，由于各条指令的功能不均衡，不利于采用先进的计算机体系结构技术（如流水技术）来提高系统的性能。

指令集功能设计

执行频率排序	80X86指令	指令执行频率
1	Load	22%
2	条件分支	20%
3	比较	16%
4	Store	12%
5	加	8%
6	与	6%
7	减	5%
8	寄存器 - 寄存器间数据移动	4%
9	调用	1%
10	返回	1%
合 计		96%

5.2 RISC技术

□ Reduced Instruction Set Computer—精简指令集计算机

✓ 形成原因

- “二八”定律，VLSI技术的发展
- 1975年，IBM的John Cocke提出了精简指令系统的设想

✓ 特点

- 指令系统简单、规则(所有指令长度均相同,在一个机器周期内完成)
- 只选取实现使用频率较高的一些简单指令
- 复杂功能通过简单指令的组合来实现
- 指令字长固定，指令格式种类少，寻址方式少
- 只有取数/存数指令访问存储器，其余指令的操作在寄存器间进行
- 寄存器数目相对较多

✓ 相关技术

- 流水线技术、超标量技术

RISC指令集功能设计原则

- 选取使用**频率**最高的指令，并补充一些最有用的指令；
- 每条指令的功能应尽可能**简单**，并在一个机器周期内完成；
- 所有**指令长度**均相同；
- 只有**load**和**store**操作指令才访问存储器，其它指令操作均在寄存器之间进行.

早期的RISC指令系统基本特征

型号	指令数	寻址方式	指令格式	通用寄存器数	主频/MHz
RISC-I	31	2	2	78	8
RISC-II	39	2	2	138	12
MIPS	55	3	4	16	4
SPARC	75	4	3	120-136	25-33
MIPSR3000	91	3	3	32	25
i860	65	3	4	32	50
Power PC	64	6	5	32	

5.3 CISC VS. RISC

	CISC	RISC
指令数目	一般多于200	一般小于100
指令格式	较多，一般多于4种	较少，一般小于4种
寻址方式	一般多于4种	一般小于4种
指令字长	不固定	等长
可访存指令	不限制	只有load/store
各种指令使用频率	相差很大	基本相当
各种指令执行时间	相差很大	多数为一个周期
程序代码长度	较短	较长
优化编译实现	很难	较容易
程序控制	微程序	组合逻辑

如何设计一个RISC/CISC 指令集，采取哪些设计策略?

主要内容

1. 指令系统概述
   - 1.1 指令系统的发展
   - 1.2 指令系统的性能要求
   - 1.3 编程语言与硬件的关系
2. 指令格式
   - 2.1 指令的一般格式
   - 2.2 指令字长
   - 2.3 指令助记符
3. 操作数与操作类型
   - 3.1 操作数类型
   - 3.2 数据在存储器中的存储方式
   - 3.3 操作类型
4. 寻址方式
   - 4.1 指令寻址
   - 4.2 数据寻址
5. CISC与RISC
   - 5.1 CISC技术
   - 5.2 RISC技术
   - 5.3 CISC与RISC的比较
6. 指令系统设计与举例
   - 6.1 简单指令集
   - 6.2 MIPS指令集
   - 6.3 x86指令集
   - 6.4 RISC-V指令集

6.1 简单指令集

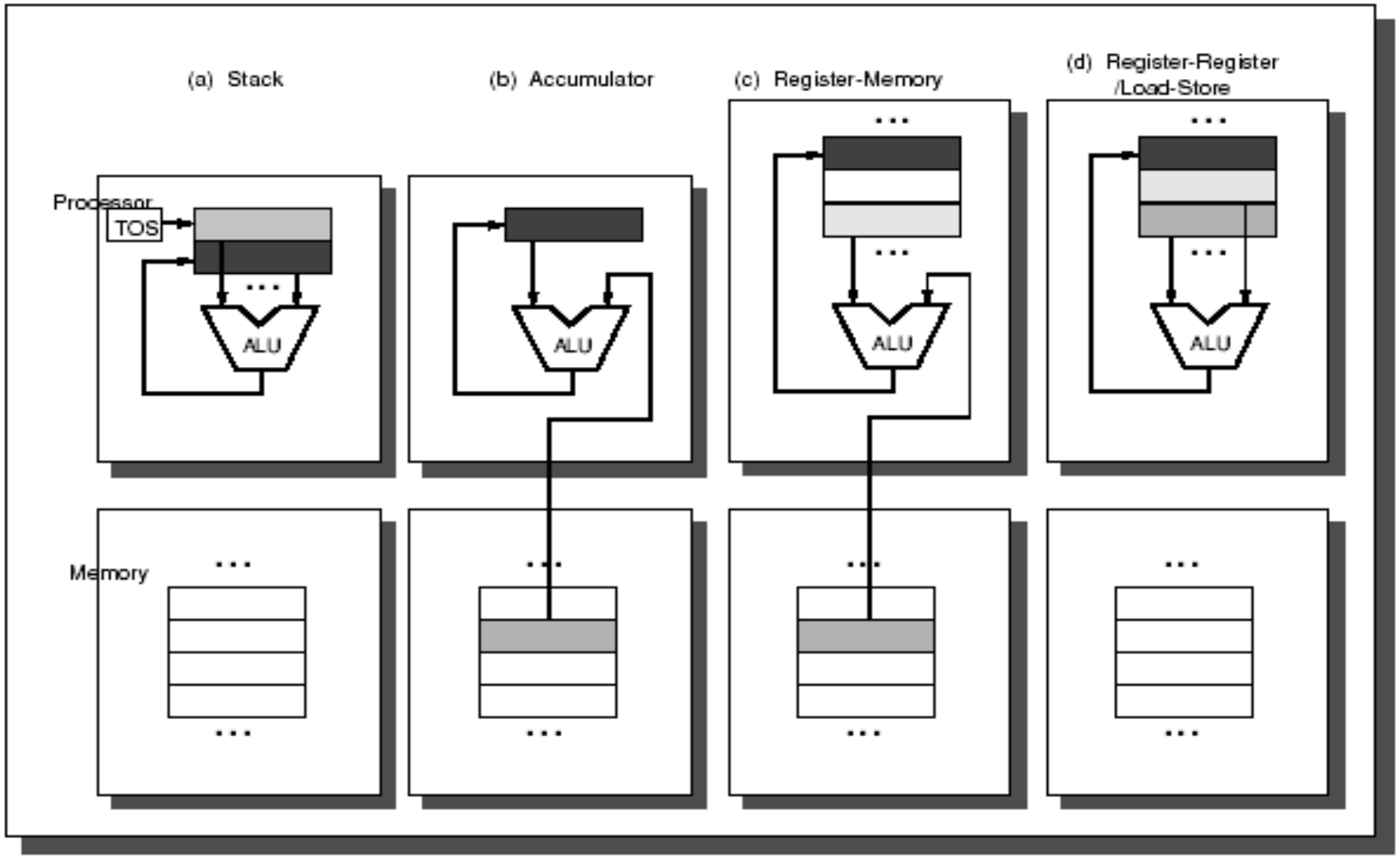
- 堆栈指令集
- 累加器指令集
- 寄存器指令集

简单指令集

□C=A+B表达式在三种指令集结构上的实现方法

堆栈	累加器	寄存器 (寄存器-存储器)	寄存器 (寄存器-寄存器)
PUSH A PUSH B ADD POP C	LOAD A ADD B Store C	LOAD R1,A ADD R3,R1,B Store R3,C	LOAD R1,A LOAD R2,B ADD R3,R1,R2 Store R3,C

ISA Classes

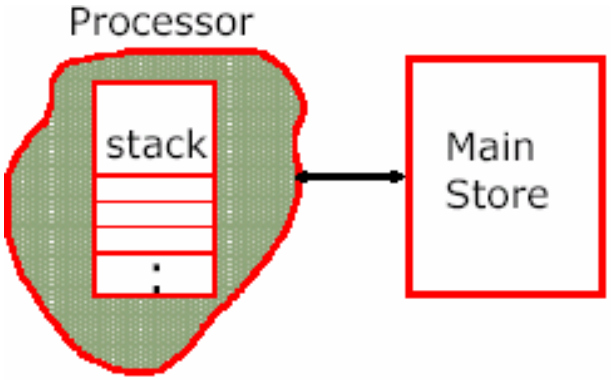

(a) Stack
(b) Accumulator
(c) Register-Memory
(d) Register-Register
/Load-Store
Processor
TOS
ALU
ALU
ALU
ALU
Memory

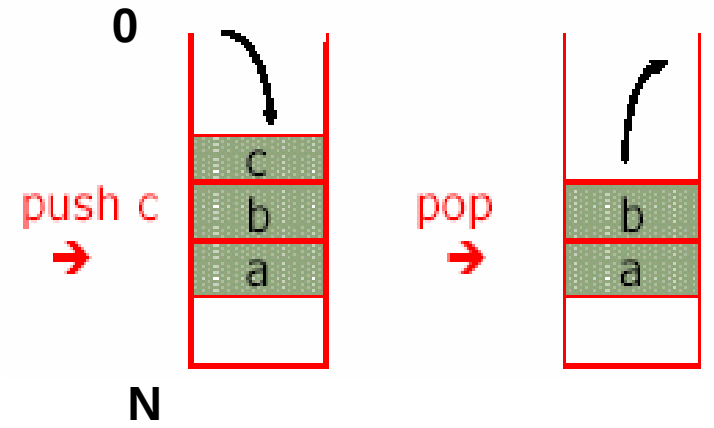
1)堆栈指令集

- 早期编译技术发挥寄存器的效率很困难，因此某些设计者完全放弃寄存器，而采用堆栈执行模型。
- 基于堆栈的操作可以有效的缩短指令字长，可以减少存储和传输的开销
  - ✓JAVA虚拟机即采用堆栈模型
  - ✓Stack在处理器中还是主存中?

1)堆栈型指令集

PUSH x	stack[sp]←M[x];sp=sp-1
POP x	M[x]←stack[sp];sp=sp+1
ADD	stack[sp+1]←stack[sp]+stack[sp+1];sp=sp+1
……	

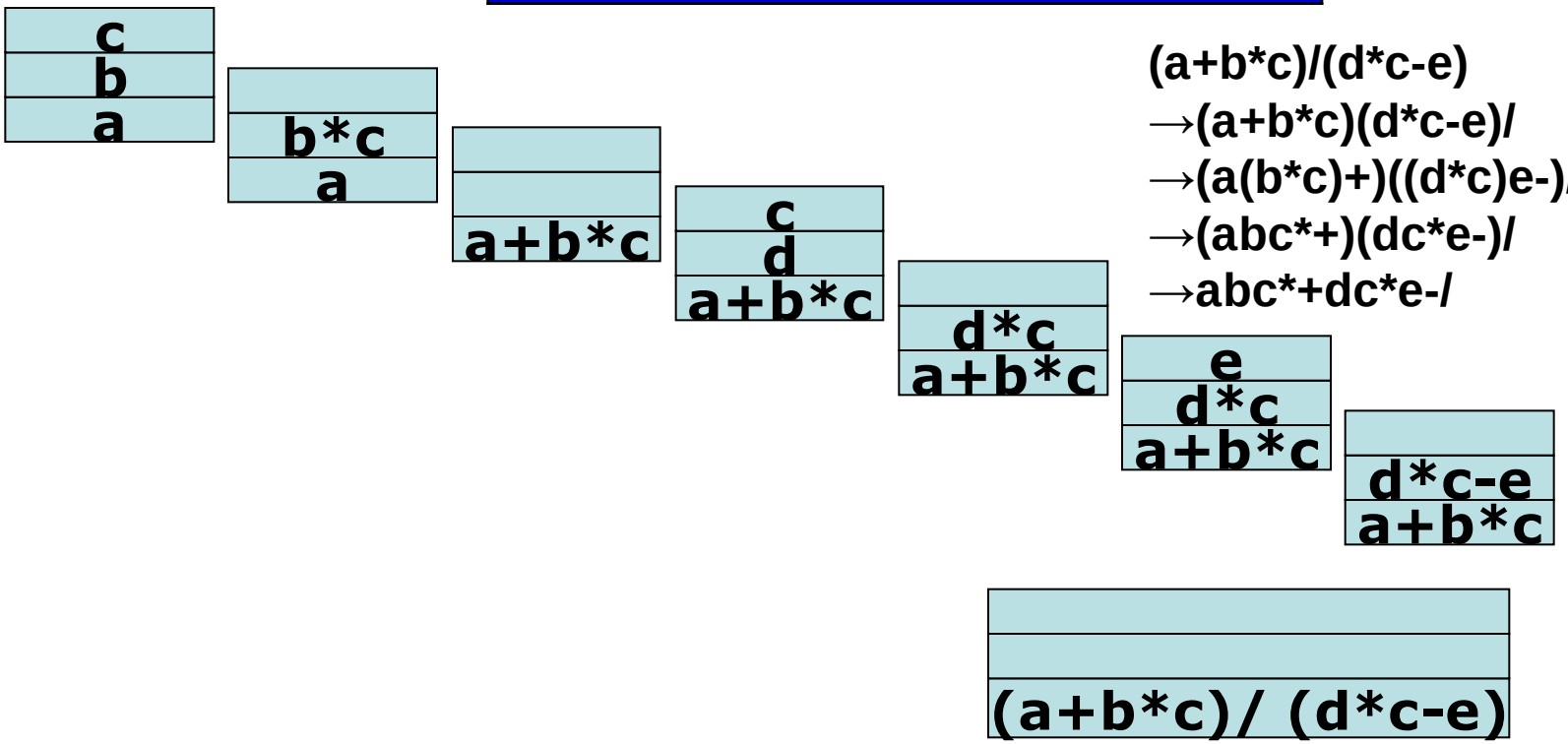




计算一个表达式

Expression (a+b*c) / (d*c-e)

Reverse Polish 后缀表达式 abc*+dc*e-/



(a+b*c)/(d*c-e)
→(a+b*c)(d*c-e)/
→(a(b*c)+)((d*c)e-)/
→(abc*+)(dc*e-)/
→abc*+dc*e-/

2）累加器指令集

- 早期硬件太昂贵，所以使用的寄存器很少。
  - ✓早期：只有一个用于算术指令的寄存器——被称为“Accumulator”，所以这种结构被称为“Accumulator Architectures”
    - 如EDSAC（第一台存储程序计算机，1949）。
  - ✓进化：采用一些专用寄存器，如数组指针、堆栈指针、乘除运算寄存器等
    - 如x86，称为“extended accumulator arch”

2）IBM 650-累加器

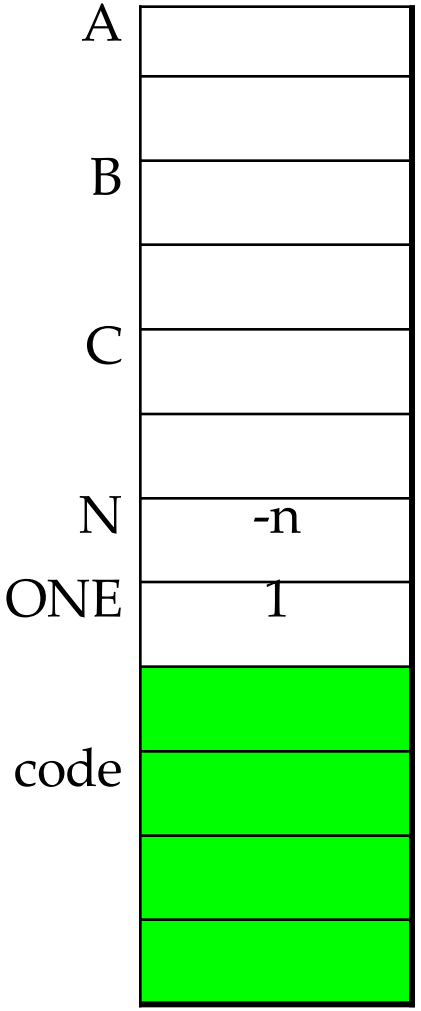
累加器型指令集结构

LOAD	X	AC←M[x]
STORE	X	M[x]←(AC)
ADD	X	AC← (AC) + M[x]
JUMP	X	PC ← x
JGE	X	if (AC) ≥0 then PC ← x
LOAD ADR	X	AC←Extract addr field(M[x])
STORE ADR	X	Extract addr field(M[x])←AC
……		

计算一个数组加

$Ci \leftarrow Ai + Bi,\ 1 \le i \le n$

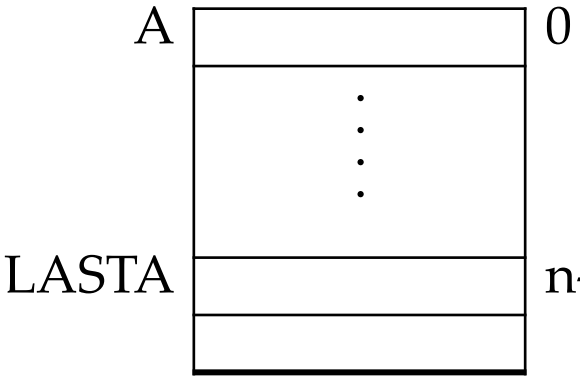
```
LOOP    LOAD        N
        JGE         DONE
        ADD         ONE
        STORE       N
F1      LOAD        A
F2      ADD         B
F3      STORE       C
        LOAD ADR    F1
        ADD         ONE
        STORE ADR   F1
        LOAD ADR    F2
        ADD         ONE
        STORE ADR   F2
        LOAD ADR    F3
        ADD         ONE
        STORE ADR   F3
        JUMP        LOOP
DONE    HLT
```

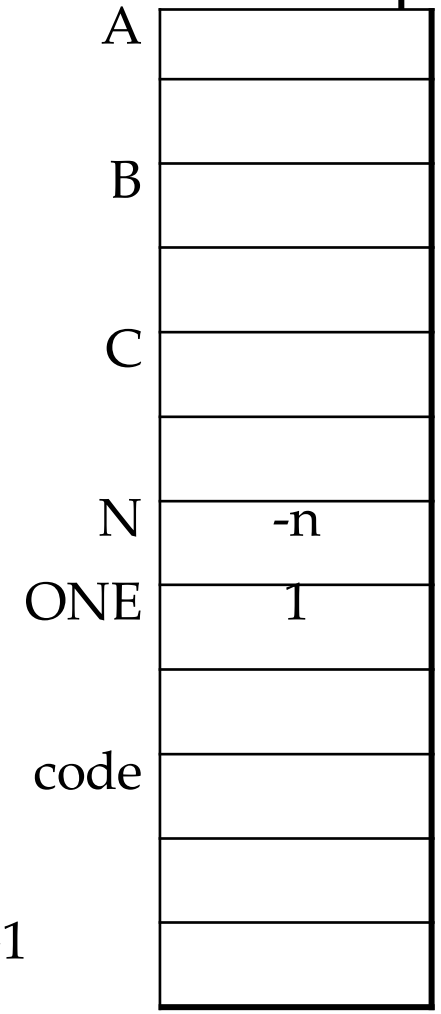


2) + index Register(IX)

……		
LOAD	x,IX	AC ← M[x+(IX)]
ADD	x,IX	AC← (AC) + M[x+(IX)]
JZi	x,IX	if (IX)=0 then PC ← x
		else IX ← (IX)+1
LOADi	x,IX	IX ← M[x]

```
        LOADi    N,IX          ; IX= -n
LOOP    JZi      DONE,IX
        LOAD     LASTA,IX
        ADD      LASTB,IX
        STORE    LASTC,IX
        JUMP     LOOP
DONE    HLT
```



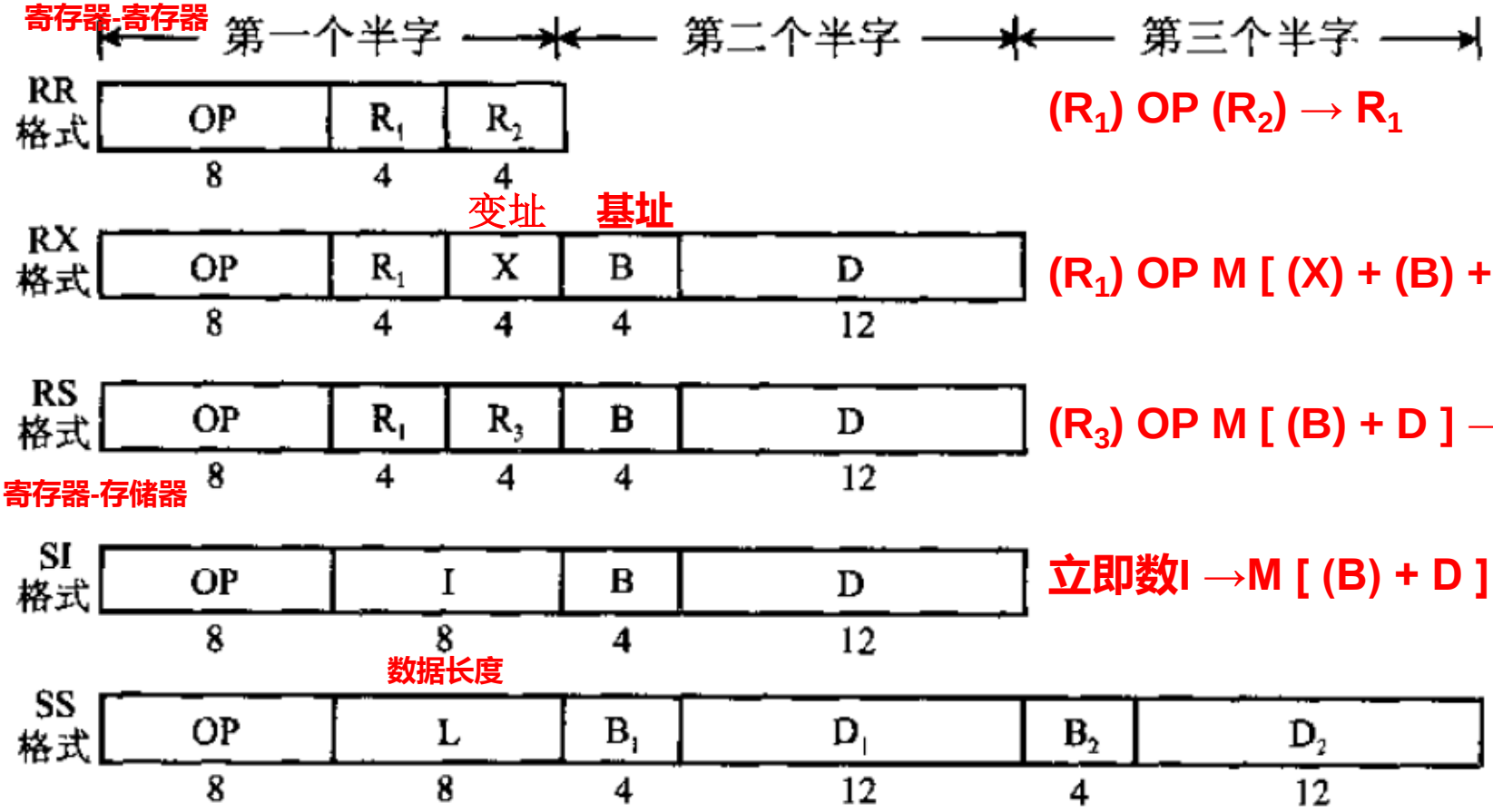


3） 典型指令系统IBM 360

- IBM 360的特点
  - ✓第一个基于寄存器的指令集结构
- IBM 360的机器属性
  - ✓32位机器，按字节寻址
  - ✓支持多种数据类型
    - 字节、半字、字、双字
    - 压缩十进制、字符串等
  - ✓16个32位通用寄存器，4个双精度（64位）浮点寄存器
    - 用户可以任选一个通用寄存器作为基址寄存器或者变址寄存器
  - ✓指令字长有16位、32位、48位三种
- IBM的指令格式

典型指令系统—IBM 360

寄存器-寄存器



6.2. MIPS指令集

- *MIPS: Microprocessor without Interlocked Pipeline Stages*

MIPS的特点

- ✓ 流行的一种RISC类型指令系统
- ✓ 机器字长有16、32、64位多种
  - 中国计算所研制的“龙芯”处理器使用64位的MIPS指令架构
- ✓ 100余条指令（Hennessy中33条）， 32个通用寄存器

MIPS的指令格式

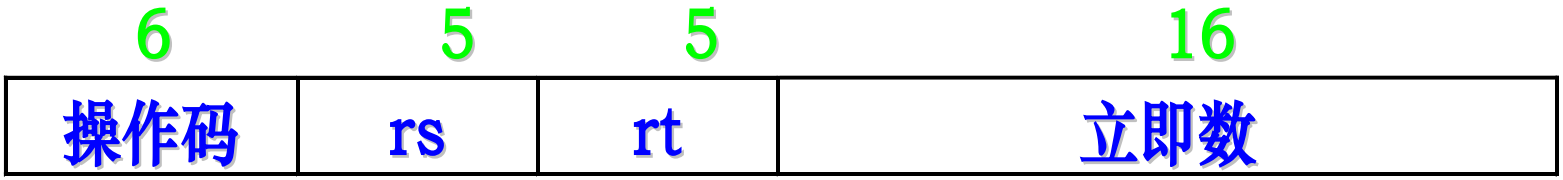
R-type *reg-reg*	op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)
I-type *reg-mm*	op(6 bits)	rs(5 bits)	rt(5 bits)	addr/immediate(16 bits)		
J-type	op(6 bits)	addr(26 bits)				

MIPS寄存器

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		
15	t7	
16	s0	callee saves
. . .	(callee must save)	
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

MIPS指令集结构：指令格式

I 类型指令



字节、半字、字的载入和存储；

rt ←rs　op　立即值。

LW R1,30(R2)

LW $s0,30($s1)

MIPS指令集结构：指令格式

R 类型指令

6	5	5	5	5	6
操作码	rs	rt	rd	shamt	Funct

1. 寄存器－寄存器 ALU 操作：rd←rs func rt
2. 函数对数据的操作进行编码：加、减、…；
3. 对特殊寄存器的读/写和移动。
4. Shamt:位移量
5. Func:函数，为 Opcode 操作某个特定变体

ADD R1，R2，R3

ADD $s0，$s1，$s2

SLT R1，R2，R3（IF RS<=R3 R1=1）

MIPS指令集结构：指令格式

中国科学技术大学
University of Science and Technology of China

J 类型指令

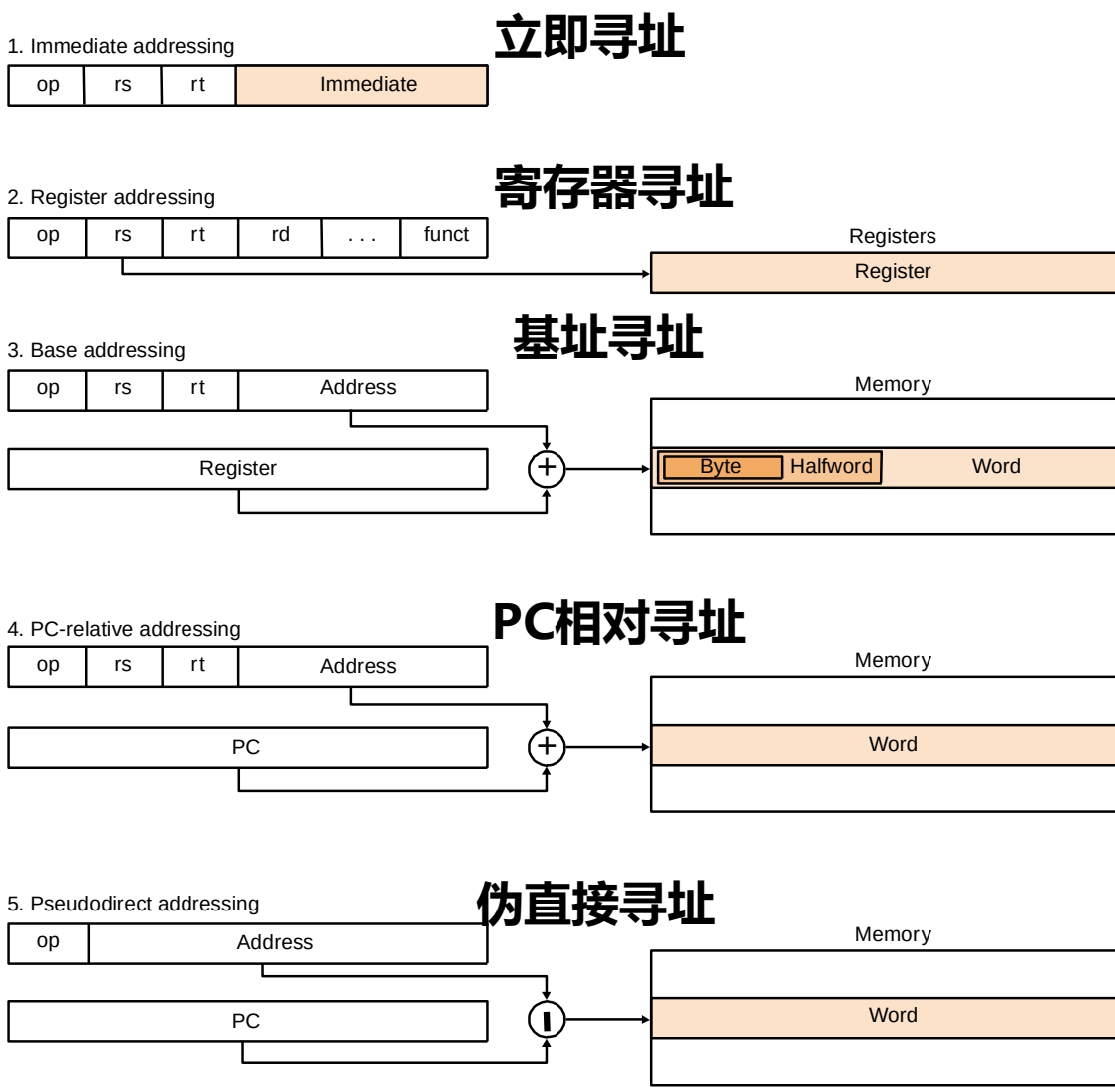
6	26
操作码	与 PC 相加的偏移量

跳转，跳转并链接，从异常（exception）处自陷和返回。

J name

MIPS寻址模式

- 立即寻址： I-type
- 寄存器寻址：R-type
- 基址寻址：I-type
- 相对寻址：I-type
- 伪直接寻址：J-type
  - ✓ 26位形式地址左移2位，与PC的高4位拼接



MIPS指令集功能

- 设计特色: *RISC*/高效/指令流水线/编译
- 算术逻辑指令、访存指令、分支跳转指令
- 算术逻辑指令Arithmetic Logical:
  - ✓Add, AddU, AddI, AddIU, Sub, SubU, And, AndI, Or, OrI, Xor, XorI, Nor, SLT, SLTU, SLTI, SLTIU
  - ✓SLL, SRL, SRA, SLLV, SRLV, SRAV

MIPS 算术指令

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$s2,$s3	$s1 = $s2 + $s3	3 operands;
subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	3 operands;
add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant;
add unsigned	addu $s1,$s2,$s3	$s1 = $s2 + $s3	3 operands;
subtract unsigned	subu $s1,$s2,$s3	$s1 = $s2 – $s3	3 operands;
add imm. unsign.	addiu $s1,$s2,100	$s1 = $s2 + 100	+ constant;
multiply	mult $s2,$s3	Hi, Lo = $s2 x $s3	64-bit signed product
multiply unsigned	multu$s2,$s3	Hi, Lo = $s2 x $s3	64-bit unsigned product
divide	div $s2,$s3	Lo = $s2 ÷ $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
divide unsigned	divu $s2,$s3	Lo = $s2 ÷ $s3, Hi = $s2 mod $s3	Unsigned quotient & remainder
Move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
Move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo

MIPS 逻辑指令

Instruction	*Example*	*Meaning*	*Comment*
and	and $s1,$s2,$s3	$s1 = $s2 & $s3	3 reg. operands; Logical AND
or	or $s1,$s2,$s3	$s1 = $s2 | $s3	3 reg. operands; Logical OR
xor	xor $s1,$s2,$s3	$s1 = $s2 Å $s3	3 reg. operands; Logical XOR
nor	nor $s1,$s2,$s3	$s1 = ~($s2 |$s3)	3 reg. operands; Logical NOR
and immediate	andi $s1,$s2,10	$s1 = $s2 & 10	Logical AND reg, constant
or immediate	ori $s1,$s2,10	$s1 = $s2 | 10	Logical OR reg, constant
xor immediate	xori $s1,$s2,10	$s1 = ~$s2 &~10	Logical XOR reg, constant
shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
shift left logical	sllv $s1,$s2,$s3	$s1 = $s2 << $s3	Shift left by variable
shift right logical	srlv $s1,$s2,$s3	$s1 = $s2 >> $s3	Shift right by variable

MIPS算术指令特点

- 设计原则*: simplicity favors regularity.*
  （简单有益于规整）
- 算术指令大都有 3 个操作数（乘除除外）
- 操作数的次序是固定的（目标操作数在前）

  示例：

  C代码： A = B + C

  MIPS代码: add $s0, $s1, $s2

MIPS算术指令

- C code: A = B + C + D;
  E = F - A;

  MIPS code:

  ```
  add $t0, $s1, $s2
  add $s0, $t0, $s3
  sub $s4, $s5, $s0
  ```

- Operands must be registers, only 32 registers provided

MIPS 数据访问指令

Instruction	*Comment*
SW $s3, 500($s4)	Store word
SH $s3, 502($s2)	Store half
SB $s2, 41($s3)	Store byte
LW $s1, 30($s2)	Load word
LH $s1, 40($s3)	Load halfword
LHU $s1, 40($s3)	Load halfword unsigned
LB $s1, 40($s3)	Load byte
LBU $s1, 40($s3)	Load byte unsigned

LUI $s1, 40 Load Upper Immediate (16 bits shifted left by 16)

访存指令Memory Access:
LB, LBU, LH, LHU, LW, LWL,LWR
SB, SH, SW, SWL, SWR

Load and Store

- Example:

  C code: A[8] = h + A[8];

  MIPS code:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```

An Example


```
swap(int v[], int k);
{ int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Assume: *k->$5*

v[0]->$4

```
swap:
    muli $2, $5, 4
    add  $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```

□MIPS 跳转指令

指令实例	指令名称	含义
J name	跳转	PC ←name + PC+4; $-2^{25} \le name \le 2^{25}$
JAL name	跳转并链接	Regs[R31] ← PC+4; PC ←name +PC+4; $-2^{25} \le name \le 2^{25}$
JR **$t0**	寄存器型跳转	PC ← Regs[$t0];
JALR **$t0**	寄存器型跳转并链接	Regs[R31]←PC+4; PC← Regs[$t0];

控制指令

- MIPS 条件分支指令:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
beqz $t0, Label
bnez $t0, Label
```

- Example: if (i==j) h = i + j;

```
bne $s0, $s1, Label
add $s3, $s0, $s1
Label: ....
```

控制指令

- MIPS unconditional branch instructions:

  J label

- Example:

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```

```
beq $s4, $s5, Lab1
add $s3, $s4, $s5
J Lab2
Lab1: sub $s3, $s4, $s5
Lab2: ...
```

练习-写出下面函数的汇编代码


```
cswap(int v[], int k, int l)
{
    int temp;
    if(v[k]!=v[l])
    {
        temp=v[k];
        v[k]=v[l];
        v[l]=temp;
    }
}
```

k -> $s5
l->$s6
v[0]->$s4

1.计算地址V[K], V[L]的地址
2.获得V[K]和V[L]
3.比较V[K]和V[L]
4. 交换V[K]和V[L]

```
muli $s2,$s5,4
add $s7,$s2,$s4
LW $s9, 0($s7)
beq $s9, $s10, Done
SW $s9, 0($s8)
Done: JR $31
```

k -> $5 l->$6 v[0]->$4


```
cswap(int v[], int k, int l)
{
    int temp;

    if(v[k]!=v[l])
    {
        temp=v[k];
        v[k]=v[l];
        v[l]=temp;
    }
}
```

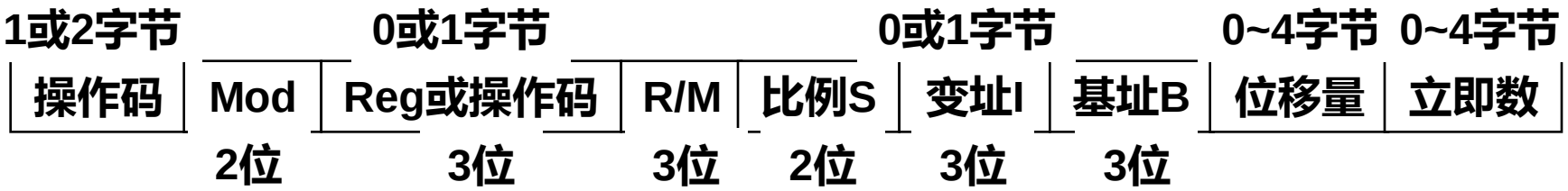
BEQ➔BEQZ指令

```
muli $s2,$s5,4
muli $s3,$s6,4
add  $s7,$s2,$4
add  $s8,$s3,$4
LW   $s9,($s7)
LW   $s10,($s8)
BEQ $s9,$s10,Label
SW   $s9,($s8)
SW   $s10,($s7)
Label: JR $31
```

```
muli $s2,$s5,4
muli $s3,$s6,4
add  $s7,$s2,$4
add  $s8,$s3,$4
LW   $s9,($s7)
LW   $s10,($s8)
Sub $s11, $s9, $s10
BEQZ $s11, Label
SW   $s9,($s8)
SW   $s10,($s7)
Label: JR $31
```

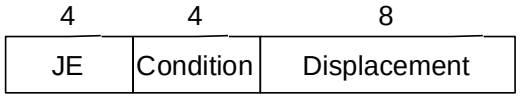
6.3 X86指令系统

- Pentium指令系统（Intel x86）
  - ✓还存活且”坚强活着”的唯一CISC类型指令系统
- 指令系统属性
  - ✓指令字长不定长，为1~12个字节，还可以带前缀
    - 前缀为可选项，作用是对其后的指令本身进行显示约定
  - ✓指令格式

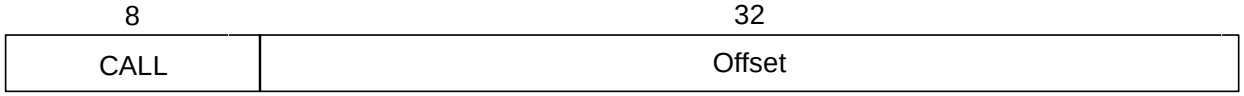


X86指令格式

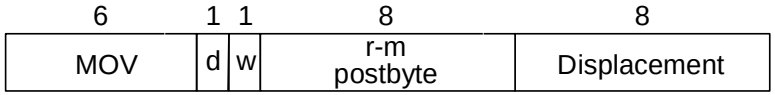
a. JE EIP + displacement



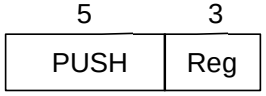
b. CALL



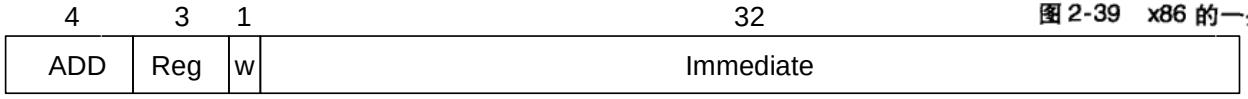
c. MOV EBX, [EDI + 45]



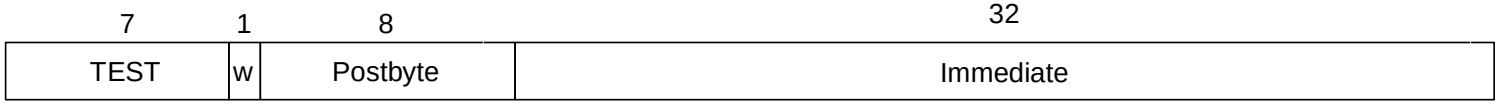
d. PUSH ESI



e. ADD EAX, #6765



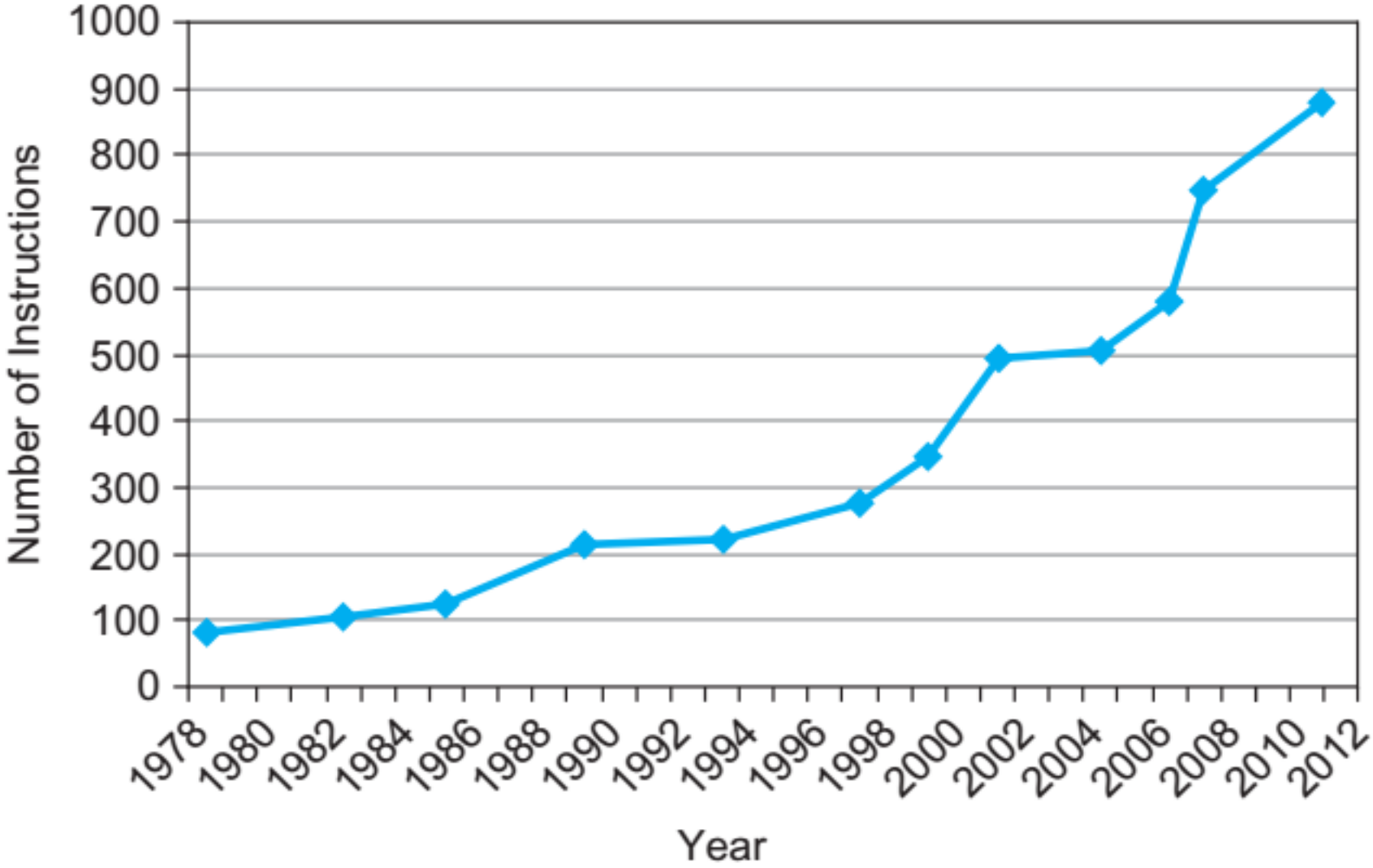
f. TEST EDX, #42



指令	功能
je name	if equal (condition code) {EIP = name; EIP - 128 <= name < EIP + 128
jmp name	EIP = name
call name	SP = SP - 4; M[SP] = EIP + 5; EIP = name;
movw EBX, [EDI + 45]	EBX = M[EDI + 45]
push ESI	SP = SP - 4; M[SP] = ESI
pop EDI	EDI = M[SP]; SP = SP + 4
add EAX, #6765	EAX = EAX + 6765
test EDX, #42	Set condition code (flags) with EDX and 42
movs l	M[EDI] = M[ESI]; EDI = EDI + 4; ESI = ESI + 4

图 2-39 x86 的一些典型指令和它们的功能

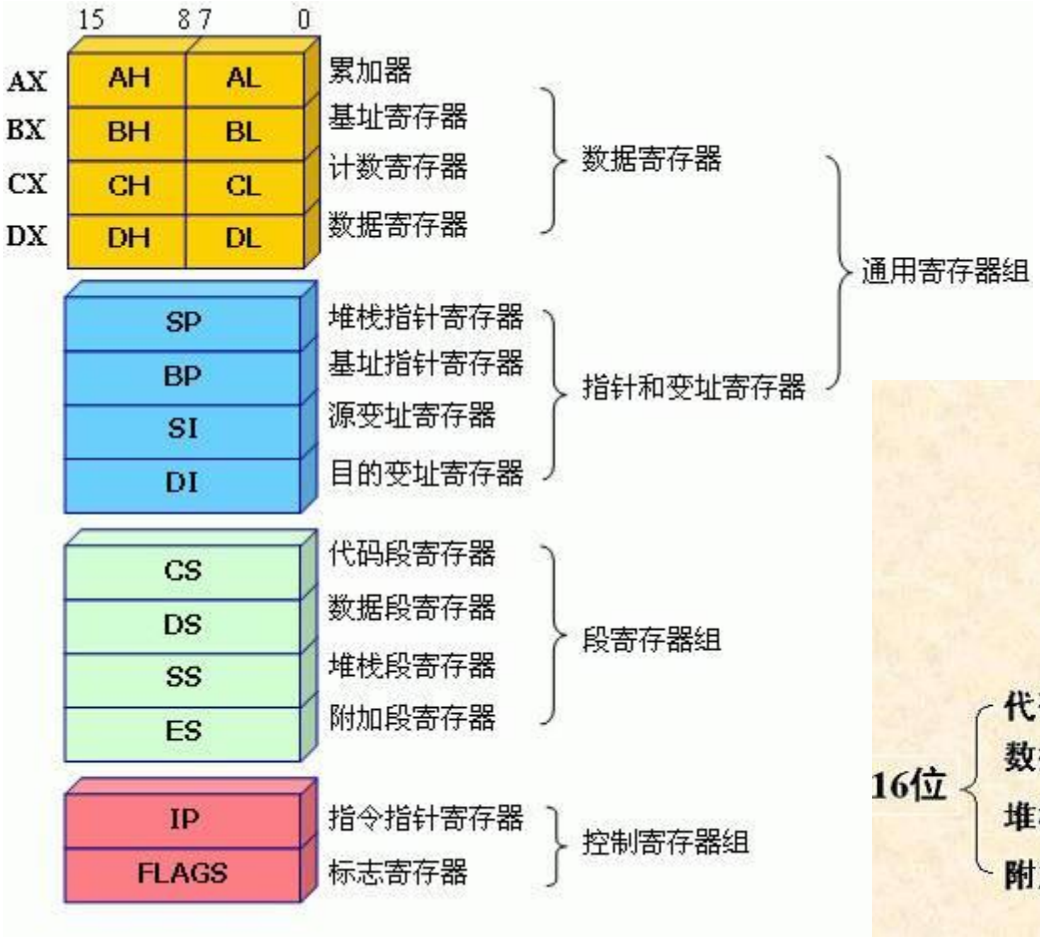
X86指令数量的增长

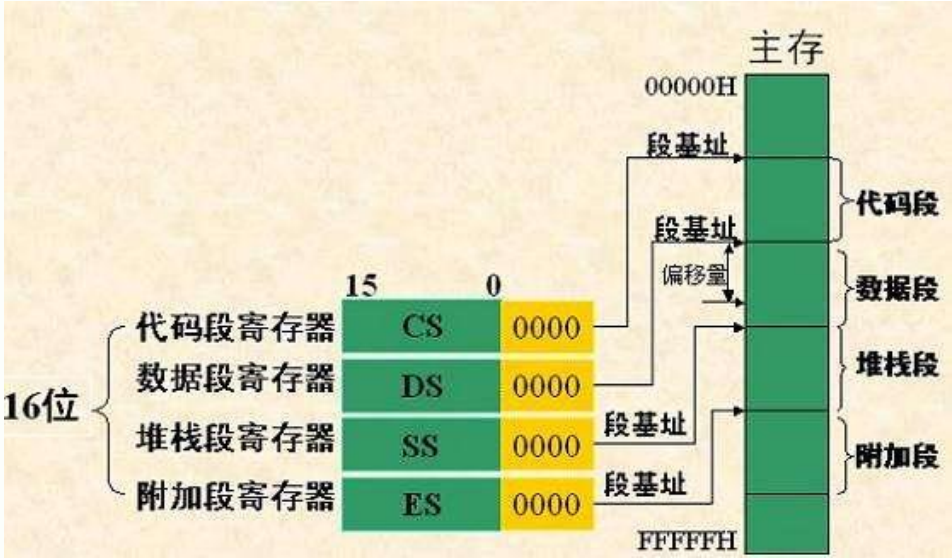

Number of Instructions
1000
900
800
700
600
500
400
300
200
100
0
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
Year

X86指令分布

Rank	80x86 instruction	Integer average (% 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%

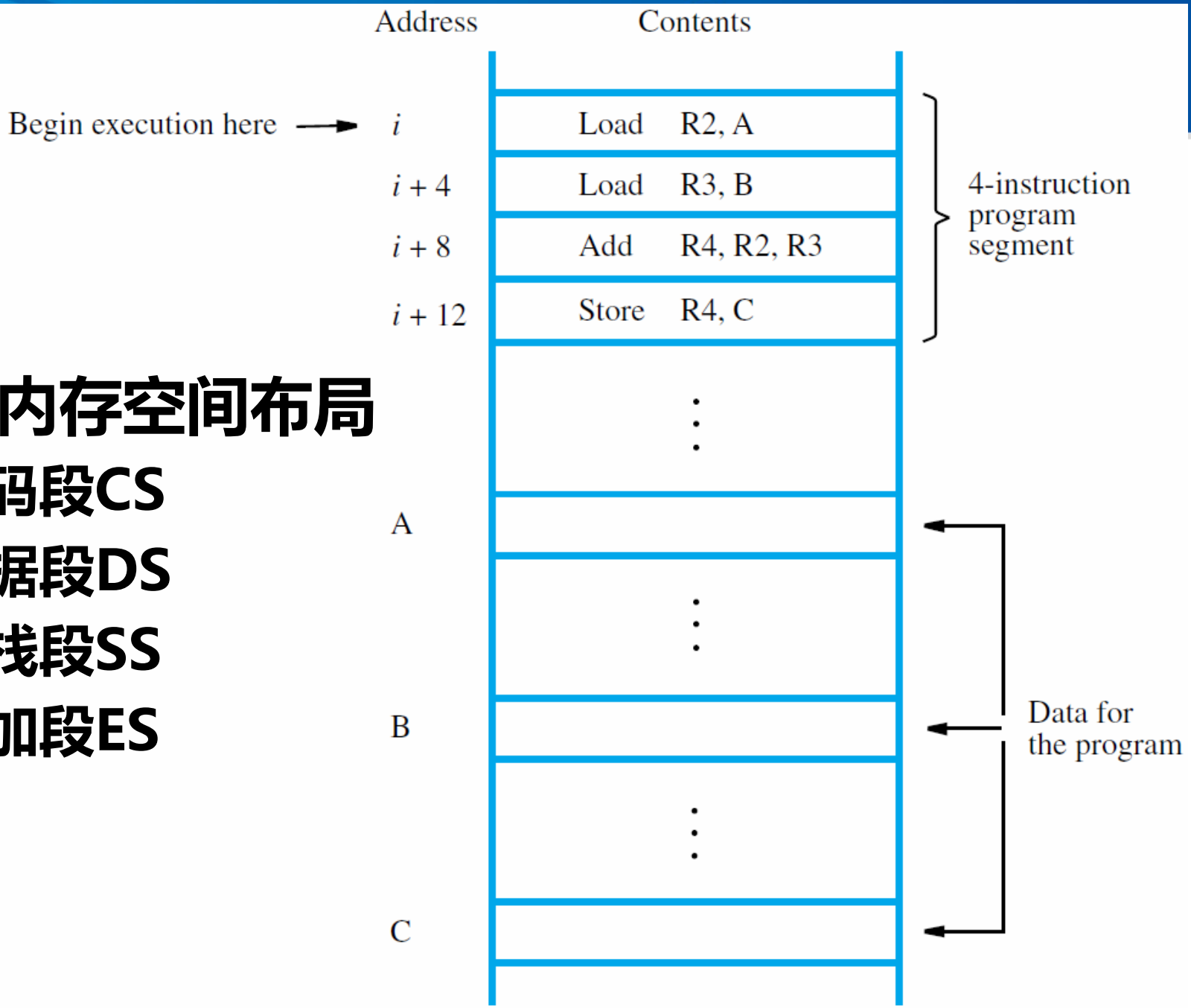
X86寄存器与段式存储



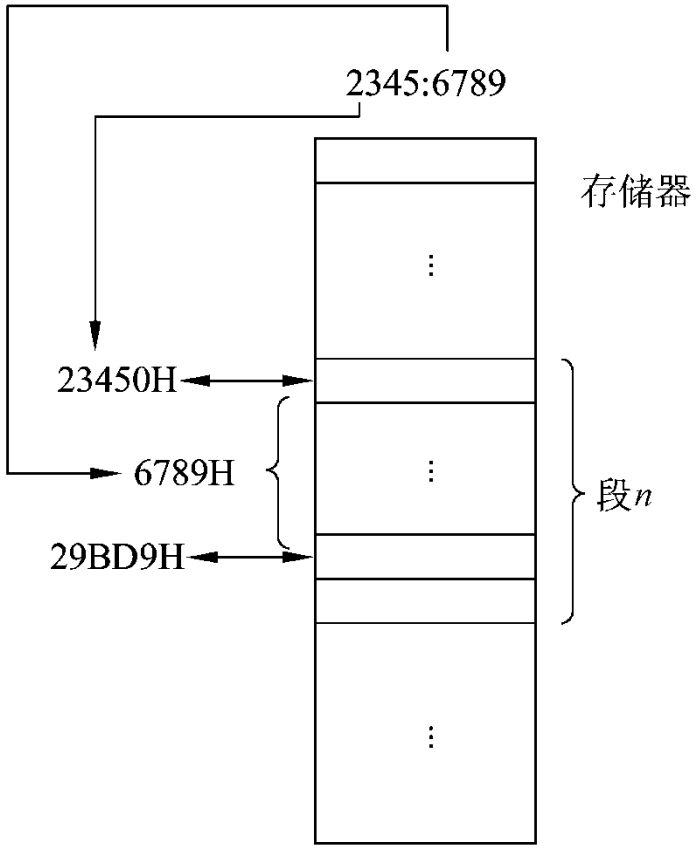


□程序内存空间布局

- ✓代码段CS
- ✓数据段DS
- ✓堆栈段SS
- ✓附加段ES



X86存储管理模式：段式

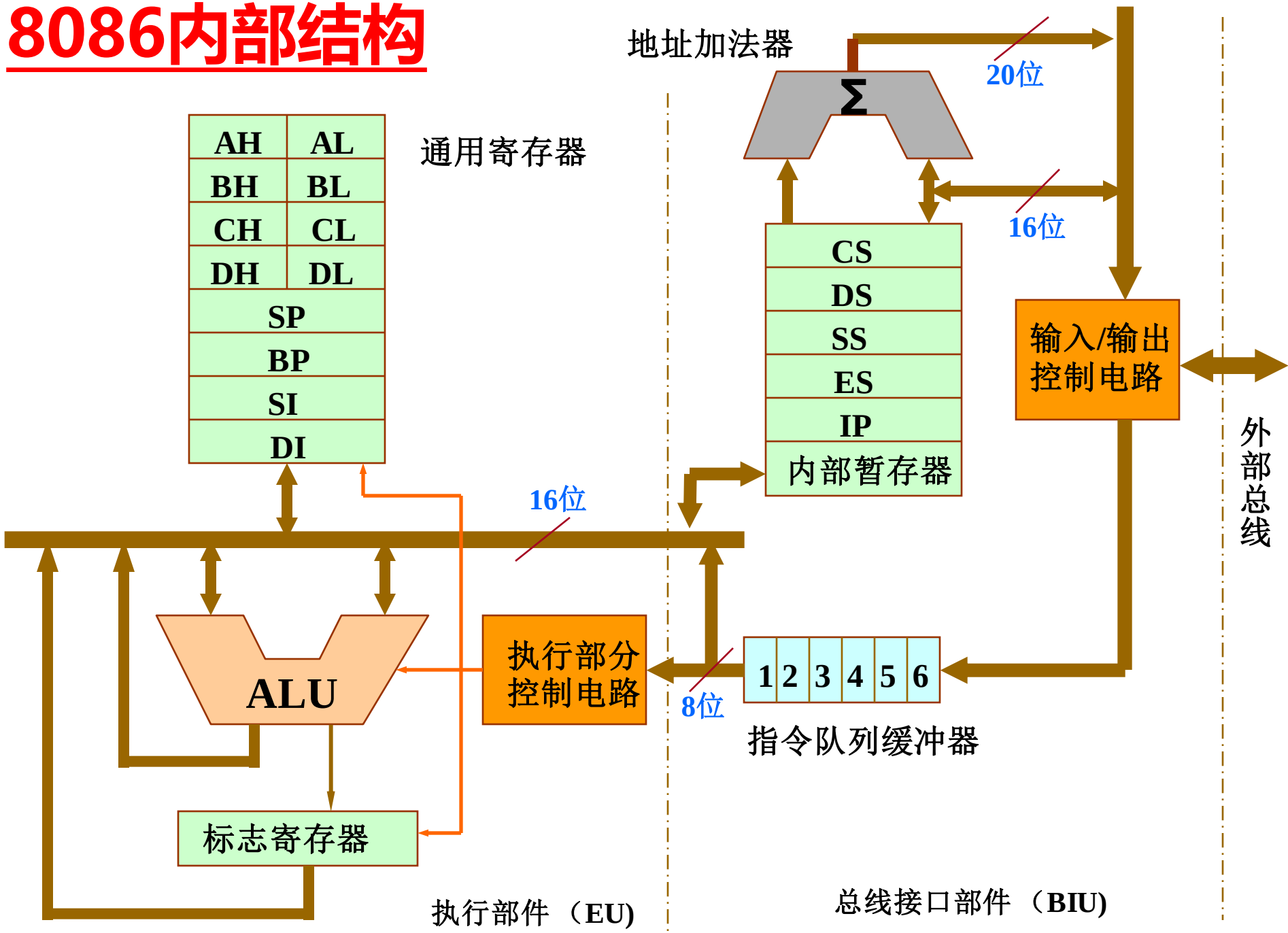



```
  2 3 4 5 0   (10H×段地址)
+   6 7 8 9   (偏移量)
-------------
  2 9 B D 9   (物理地址)
```

```
1398:0100 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0110 00 00 00 00 00 00 00 00-00 00 00 00 34 00 87 13 ............4...
1398:0120 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0130 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0140 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0150 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0160 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0170 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

- 内存管理：20位地址
- 存储模型
  - 段基址寄存器16位（64K）
  - 数据段（ds）、代码段（cs）、堆栈段（ss）、扩展段（es）
- 16位基址:16位偏移

8086内部结构


地址加法器
Σ
20位
16位
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
通用寄存器
CS
DS
SS
ES
IP
内部暂存器
输入/输出
控制电路
外部总线
16位
ALU
执行部分
控制电路
1 2 3 4 5 6
8位
指令队列缓冲器
标志寄存器
执行部件（EU)
总线接口部件（BIU)

8086标志寄存器：系统当前状态与控制

堆栈段的段基址由段寄存器SS指定，栈顶地址由堆栈指示器SP指定。
8086系统中的堆栈是一个向上生长型的，堆栈地址由高向低变化。

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
				OF	DF	IF	TF	SF	ZF		AF		PF		CF	8086/8088

1、OF：溢出标志。
2、DF：方向标志。
3、IF：中断允许标志。
4、TF: 陷阱标志。
5、SF：符号标志。
6、ZF：零标志。
7、AF：辅助进位标志。。
8、PF：奇偶标志
9、CF：进位/借位标志。


```
C:\Users\Administrator>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B2D  ES=0B2D  SS=0B2D  CS=0B2D  IP=0100   NV UP EI PL NZ NA PO NC
0B2D:0100 7438          JZ      013A
-
```

- 指令：机器指令助记符
- 伪指令
  - ✓ DW 定义字(2字节)
  - ✓ PROC 定义过程
  - ✓ ENDP 过程结束
  - ✓ SEGMENT 定义段
  - ✓ ASSUME 建立段寄存器寻址
  - ✓ ENDS 段结束
  - ✓ END 程序结束
- 系统调用（system calls）：BIOS，DOS
  - ✓ 显示、磁盘、文件、打印机、时间

```
data SEGMENT '数据段，编程者可以把数据都放到这个段里
….数据部分
'数据格式是： 标识符 db/dw 数据。
data ENDS'数据段结束处。

edata SEGMENT '附加数据段，编程者可以把数据都放到这个段里
….附加数据部分
edata ENDS'附加数据段结束处。

code SEGMENT'代码段，实际的程序都是放这个段里。
ASSUME CS:code,DS:data,ES:edata '告诉编译程序，data段是数据段DS，code段是代码段CS
start:MOV AX,data '前面的start表示一个标识位，后面用到该位，如果用不到，就可以不加
MOV DS,AX '这一句与上一行共同组成把data赋值给DS。段寄存器.
MOV AX,edata
MOV ES,AX '与前一句共同组成edata->ES
…….程序部分
MOV AX,4C00h'程序退出，该句内存由下一行决定。退出时，要求ah必须是4c。
INT 21h       INT 21H  指令自动转入中断子程序
code ENDS'代码段结束。
END start'整个程序结束，并且程序执行时由start那个位置开始执行。
```

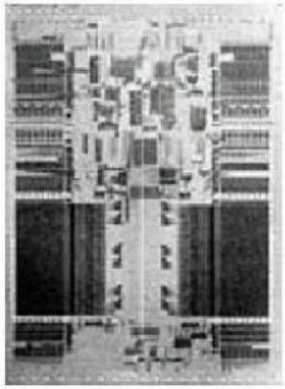
6.4 RISC-V

- RISC-V@UCB，2010~2014 Prof.Krste Asanovic、Prof. David Patterson
- RISC-V：40 integer instructions
- Rocket Chip Generator，开源处理器
- Rocket：标量
  - ✓ 机器字长64 位；
  - ✓ 5 级pipeline，采用单发射；
  - ✓ 支持虚拟内存，可以兼容的移植开源操作系统；
  - ✓ 分支推断缓存（Branch Prediction Buff）、分支历史表（Branch History Table）、返回栈（Return Address Stack）
  - ✓ Rocket Custom Coprocessor Interface (RoCC)
- BOOM（Berkeley Out-of-Order Machine）：超标量
  - ✓ 机器字长为64 位，支持指令集为RV64G；
  - ✓ 6 级流水线，乱序发射；
- AHB-Lite buses
- ISA simulator

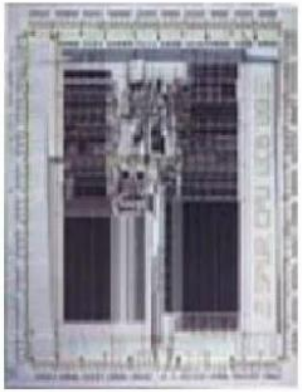
History for RISC-V

RISC-I
1981

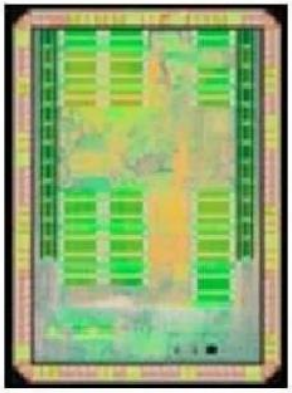
RISC-II
1983



RISC-III (SOAR)
1984



RTSC-IV (SPUR)
1988



RTSC-V
2013

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

RISC-V Registers

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

FIGURE 2.14 RISC-V register conventions. This information is also found in Column 2 of the RISC-V Reference Data Card at the front of this book.

- x0: 常数0
- x1: 返回地址
- x2: 栈指针
- x3: 全局指针
- x4: 线程指针
- x5 – x7, x28 – x31: 临时变量
- x8: 帧指针
- x9, x18 – x27: 其值需要保存的寄存器
- x10 – x11: 函数参数/结果
- x12 – x17: 函数参数

寄存器操作数的例子

- C code:

```
f = (g + h) - (i + j);
```

  - ✓f, …, j保存在x19, x20, …, x23中

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

RISC-V 存储器

- 主存储器用于存储复合数据
  - ✓ 数组、结构、动态数据
- 目的：为了进行算术运算操作
  - ✓ 把数值从存储器取到寄存器
  - ✓ 把结果从寄存器存到存储器
- 内存按字节编址
  - ✓ 每个地址对应一个8位字节
- RISC-V采用小端模式
  - ✓ 低位字节位于字中的低地址
  - ✓ 对比大端模式：高位字节位于低地址
- RISC-V不要求字在内存中对齐
  - ✓ 和其他一些ISA不同

存储器操作数的例子

- C code:

```
A[12] = h + A[8];
```

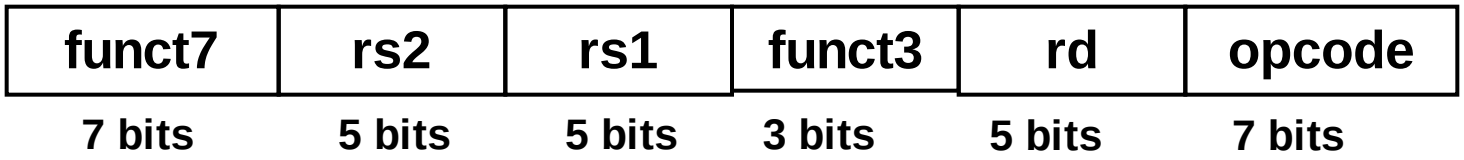
  - ✓h保存在x21中，A的基址保存在x22中

- Compiled RISC-V code:

  - ✓下标8对应的偏移量是64（**每个双字有8个字节**）

```
ld      x9, 64(x22)
add     x9, x21, x9
sd      x9, 96(x22)
```

RISC-V R型指令



- **Instruction fields**
  - ✓**opcode:**操作码
  - ✓**rd:**目标寄存器号
  - ✓**funct3:** 3位功能码（扩展操作码）
  - ✓**rs1:**第一个源操作数寄存器号
  - ✓**rs2:**第二个源操作数寄存器号
  - ✓**funct7:** 7位功能码（扩展操作码）

R型指令的例子

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

```
add x9,x20,x21
```

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

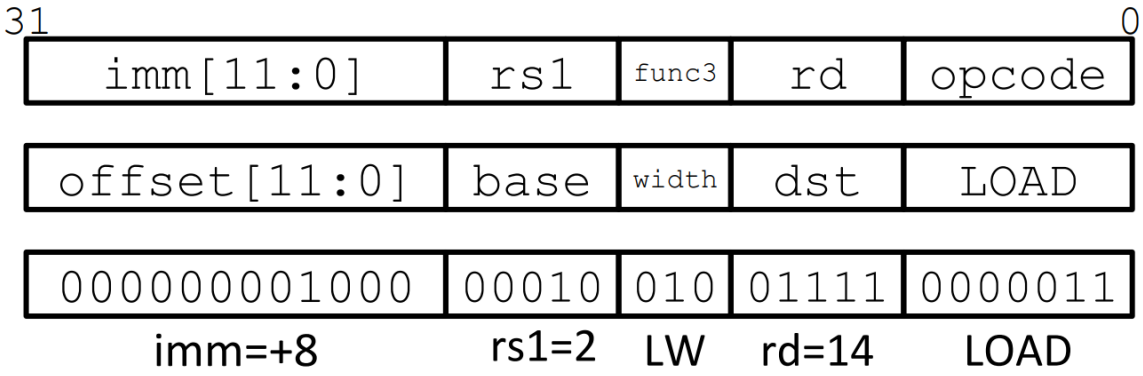
$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

RISC-V I型指令

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- 立即数算术运算和取数指令
  - ✓ **rs1:**源或基地址寄存器号
  - ✓ **immediate:**常数操作数，或是加在基地址上的偏移量
- **设计原则：好的设计需要好的折中**
  - ✓ 不同的格式会使指令解码变复杂，但能一致使用32位的指令
  - ✓ 保持不同格式尽可能相似

- **`lw x14, 8(x2)`**

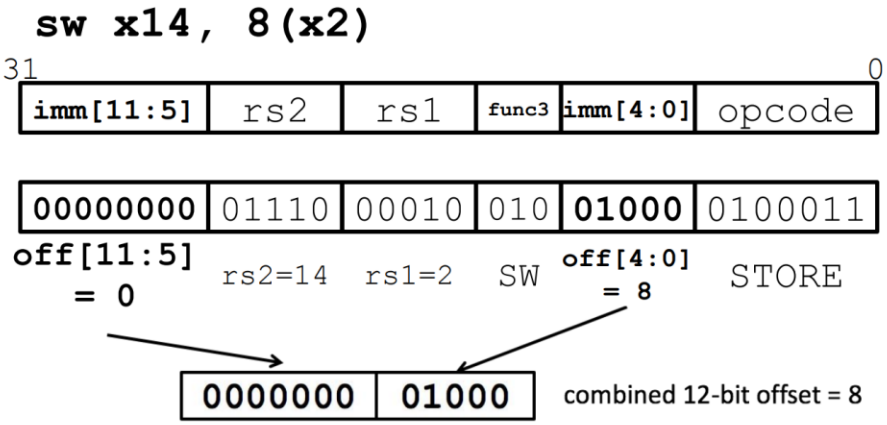


RISC-V S型指令

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- 用于存数指令的另一种立即数格式
  - ✓ rs1：基地址寄存器号
  - ✓ rs2：源操作数寄存器号
  - ✓ immediate：加在基地址上的偏移量
  - ✓ **12位立即数拆分为7位和5位两个字段，是为了保持rs1和rs2字段位置不变**

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

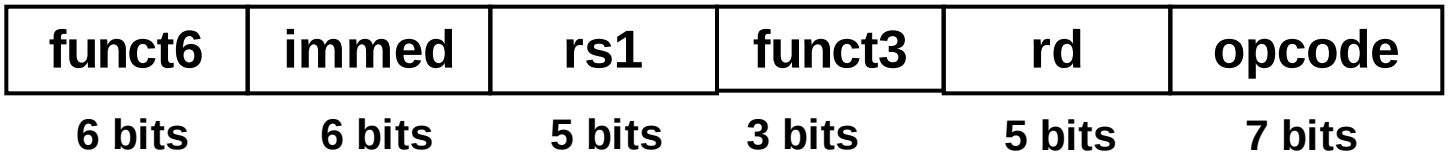


口按位操作的指令

操作	C	Java	RISC-V
左移	<<	<<	slli
右移	>>	>>>	srli
按位与	&	&	and，andi
按位或	|	|	or，ori
按位异或	^	^	xor，xori
按位取反	~	~	Xori(全1)

- **用于对一个字提取或插入多组数位**

移位运算



- immed：移动的位数（不能大于63位）
- 逻辑左移
  - ✓左移并以0填充空位
  - ✓slli指令左移*i*位相当于乘以2^i
- 逻辑右移
  - ✓右移并以0填充空位
  - ✓srli指令右移*i*位相当于除以2^i（仅适用于无符号数）

与运算

- 用于对一个字中的某些位进行掩码
  - ✓选出某些位，把其他位清0

```
and  x9,x10,x11
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

或运算

- 用于计入字中的某些位
  - ✓把某些位置1，保持其他位不变

```
or x9,x10,x11
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

异或运算

◘差异运算

✓把某些位置1，保持其他位不变

```
xor x9,x10,x12   // # 按位取反运算
```

x10	00000000	00000000	00000000	00000000	00000000	00000000	00001101	11000000
x12	11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
x9	11111111	11111111	11111111	11111111	11111111	11111111	11110010	0111111

条件操作（分支控制指令）

- 如果条件为真则转到一个有标签的语句
  - ✓否则按顺序执行

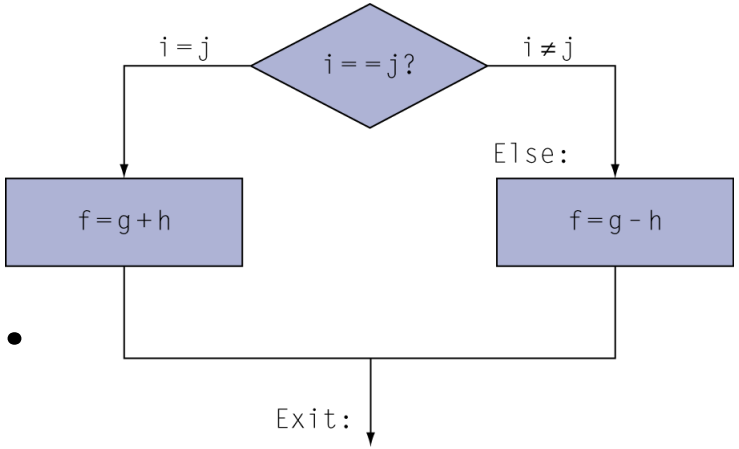
- **`beq rs1, rs2, L1`**
  - **✓如果(rs1 == rs2)则转到标签为L1的语句**
- **`bne rs1, rs2, L1`**
  - **✓如果(rs1 != rs2)则转到标签为L1的语句**

编译if语句

C code:

```
if (i==j) f = g+h;
else f = g-h;
```

- ✓f, g, … 保存在 x19, x20, …



编译的 RISC-V code:

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0,x0,Exit // 无条件跳转
Else:   sub x19, x20, x21
Exit:   …
```

汇编器计算地址

编译循环语句

- C code:

```
while (save[i] == k) i += 1;
```

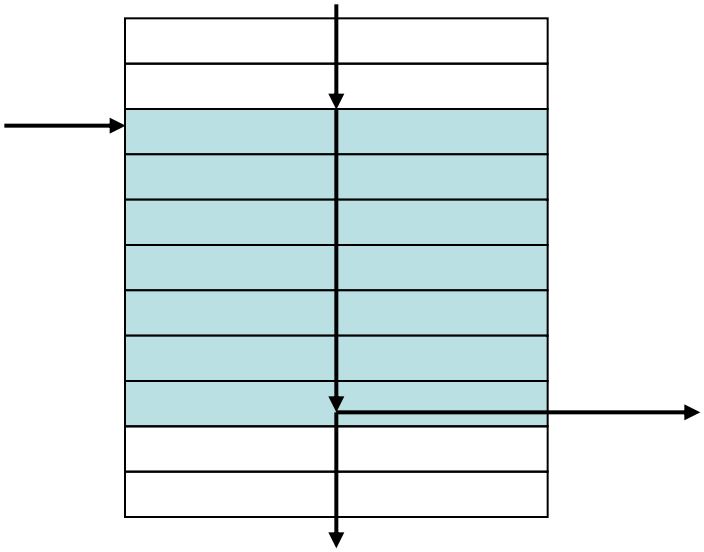
  - ✓i在x22中，k在x24中，save的地址在x25中

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

基本块Basic Blocks

- 基本块是指令序列
  - ✓没有嵌入分支（除非在末尾）
  - ✓没有分支目标（除非在开头）



- 编译器可以识别基本块以进行优化
- 先进的处理器能够加速基本块的执行

更多的条件操作

- `blt rs1, rs2, L1`
  - ✓ if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - ✓ if (rs1 >= rs2) branch to instruction labeled L1
- Example
  - ✓ if (a > b) a += 1;
  - ✓ a in x22, b in x23

```
bge  x23, x22, Exit      // branch if b >= a
addi x22, x22, 1
Exit:
```

有符号与无符号

- 有符号数比较：blt, bge
- 无符号数比较：bltu, bgeu
- **Example**
  - ✓**x22 = 1111 1111 1111 1111 1111 1111 1111 1111**
  - ✓**x23 = 0000 0000 0000 0000 0000 0000 0000 0001**
  - ✓**x22 < x23 // signed**
    - **–1 < +1**
  - ✓**x22 > x23 // unsigned**
    - **+4,294,967,295 > +1**

字节/半字/字操作

- RISC-V字节/半字/字的取数/存数
  - ✓从内存读一个字节/半字/字：符号扩展为64位存入rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - ✓从内存读一个无符号的字节/半字/字：零扩展为64位存入rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - ✓将一个字节/半字/字存入内存：保存最右端的8/16/32位
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

U类指令-加载32-bit 常量

- 常量一般较小,12位立即数就够用,对于偶尔用到的32位常量,如何处理?
- lui rd, constant
  - ✓ 复制20位常量到rd[31:12]，符号扩展rd[63:32]
  - ✓ 将rd[11:0]清0

```
lui x19, 976  // 0x003D0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

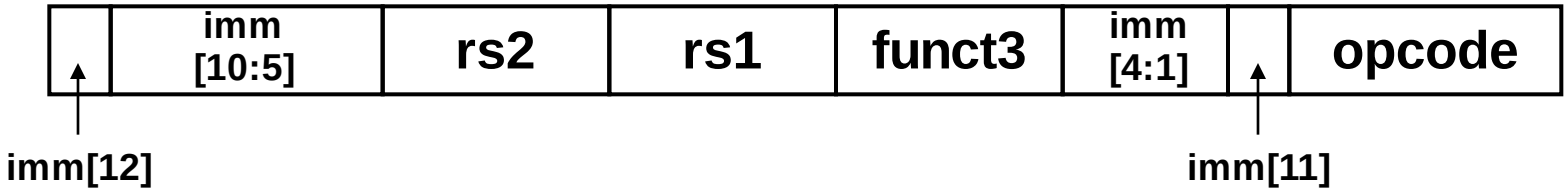
```
addi x19,x19,128  // 0x500
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

SB类指令-分支寻址

- 分支指令指定
  - ✓操作码、两个寄存器、目标地址
- 多数分支是近分支
  - ✓向前或向后
- SB格式：



bne x10, x11, 2000 // if x10 != x11, go to location 2000ten = 0111 1101 0000

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

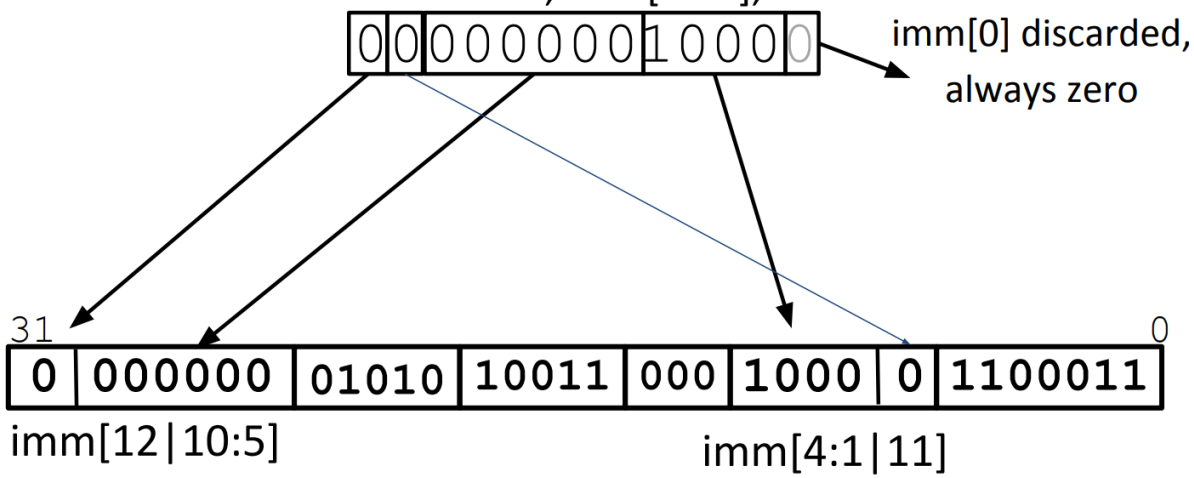
bne 条件分支

SB类指令-分支寻址

口分支为偶地址，故最低位不存

```
beq  x19,x10,offset = 16 bytes
```

13-bit immediate, imm[12:0], with value 16



注意立即数的存放方式

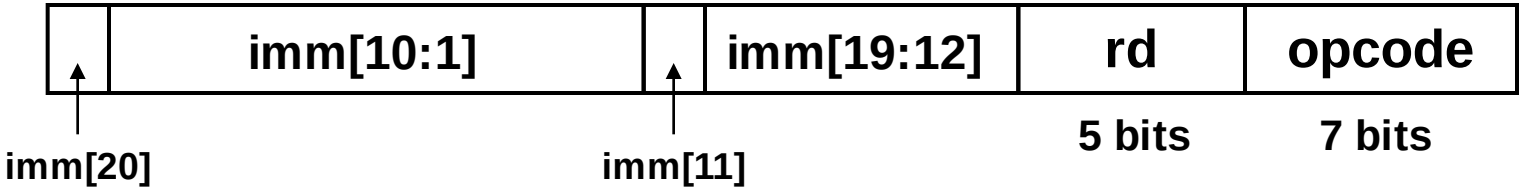
设计原则：好的设计需要好的折中

不同的格式会使指令解码变复杂，但能一致使用32位的指令；保持不同格式尽可能相似

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

UJ类指令-跳转寻址

- jal的目标地址使用20位的立即数来实现更大范围的跳转
- UJ（无条件跳转）格式：



注意立即数的存放方式

设计原则：好的设计需要好的折中

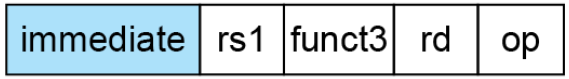
不同的格式会使指令解码变复杂，但能一致使用32位的指令；保持不同格式尽可能相似

jal x0, 2000 // go to location 2000ten = 0111 1101 0000

0	1111101000	0	00000000	00000	1101111
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

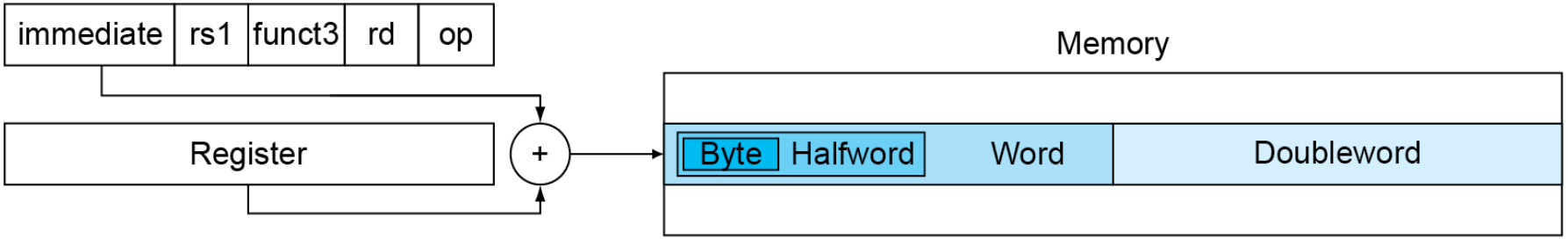
RISC-V寻址模式总结

1. Immediate addressing

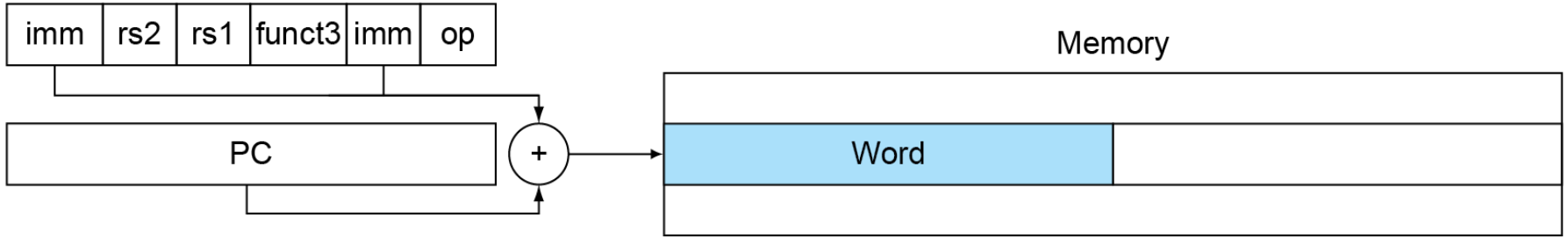


2. Register addressing

3. Base addressing



4. PC-relative addressing



几点说明

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

- S类：
  - ✓ 12位立即数拆分为7位和5位两个字段，保持rs1和rs2字段位置不变
- SB类：
  - ✓ 跳转最低位为0（只能跳转到偶地址），可表示13位地址
  - ✓ 保持[10:5][4:1]与S类型一致
  - ✓ 最高位为符号位，与S类一致
- UJ类
  - ✓ 跳转最低位为0（只能跳转到偶地址），可表示21位地址
  - ✓ 尽可能多的与其他类型指令保持一致：[10:5]与SB类一致，[19:12]与U类一致
  - ✓ 符号位最高位，与其他类型一致

RISC-V指令编码格式总结

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	
R-type	funct7	rs2	rs1	funct3	
I-type	immediate[11:0]		rs1	funct3	
S-type	immed[11:5]	rs2	rs1	funct3	im
SB-type	immed[12,10:5]	rs2	rs1	funct3	imm
UJ-type	immediate[20,10:1,11,19:12]				
U-type	immediate[31:12]				

R-type Instructions	funct7	rs2	rs1	funct3	
add (add)	0000000	00011	00010	000	00
sub (sub)	0100000	00011	00010	000	00
I-type Instructions	**immediate**		**rs1**	**funct3**	
addi (add immediate)	001111101000		00010	000	00
ld (load doubleword)	001111101000		00010	011	00
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**im -i**
sd (store doubleword)	0011111	00001	00010	011	01

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	iw	0000011	010	n.a.
	id	0000011	011	n.a.
	ibu	0000011	100	n.a.
	ihu	0000011	101	n.a.
	iwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

FIGURE 2.18 RISC-V instruction encoding.

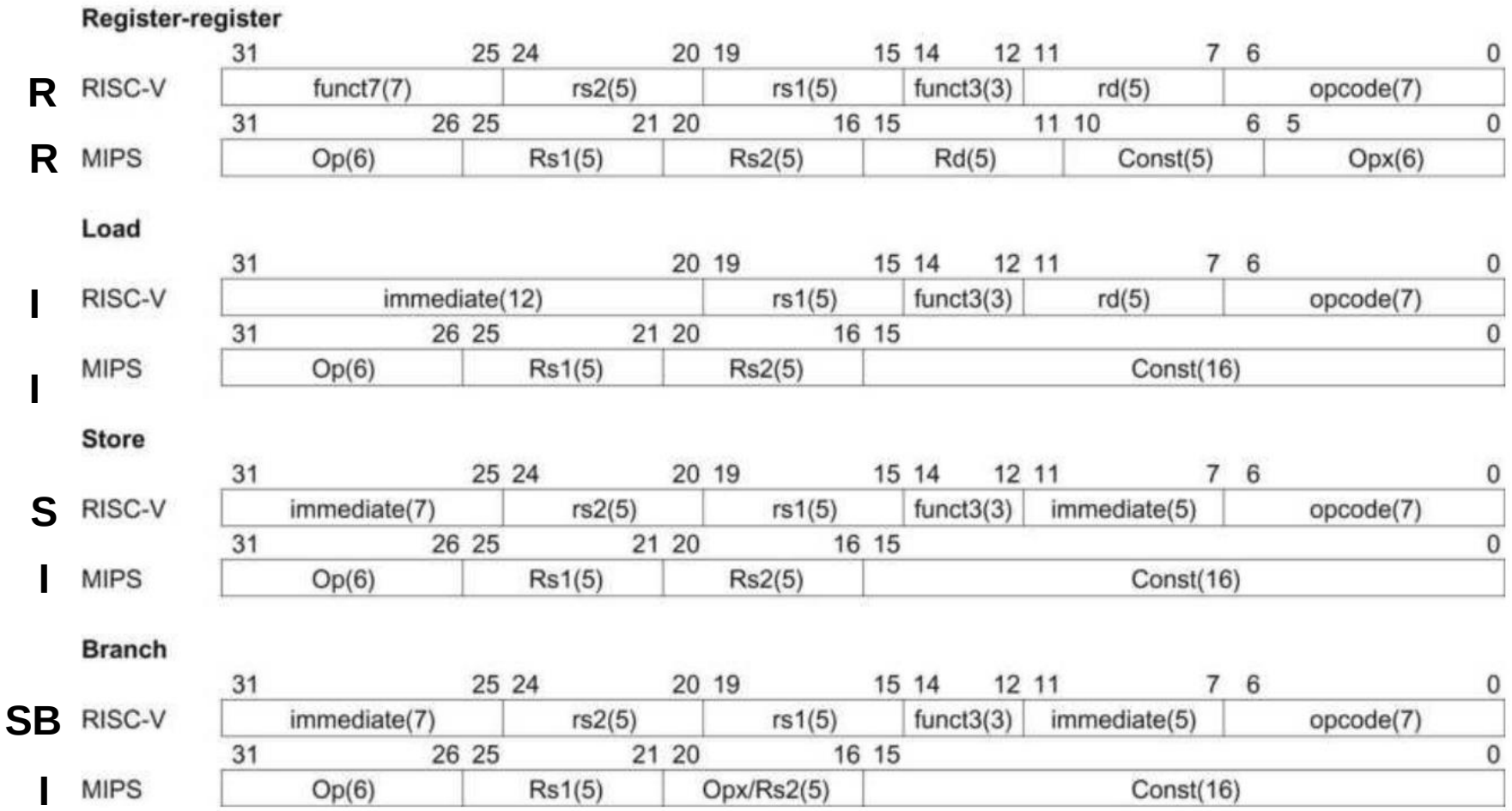
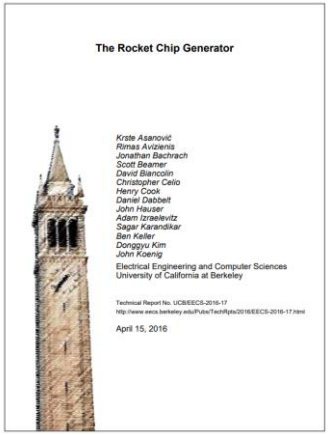


FIGURE 2.29 **Instruction formats of RISC-V and MIPS.**

基于RISC-V指令集的处理器项目 Rocket Chip Generator 和BOOM

由UCB提出的两个开源处理器项目

- Rocket Chip Generator，开源处理器
- Rocket：标量
  - ✓ 机器字长64 位；
  - ✓ 5 级pipeline，采用单发射；
  - ✓ 支持虚拟内存，可以兼容的移植开源操作系统；
  - ✓ 分支推断缓存（Branch Prediction Buff）、分支历史表（Branch History Table）、返回栈（Return Address Stack）
  - ✓ Rocket Custom Coprocessor Interface (RoCC)
- BOOM（Berkeley Out-of-Order Machine）：超标量
  - ✓ 机器字长为64 位，支持指令集为RV64G；
  - ✓ 6 级流水线，乱序发射；


The Rocket Chip Generator

Krste Asanović
Rimas Avizienis
Jonathan Bachrach
Scott Beamer
David Biancolin
Christopher Celio
Henry Cook
Daniel Dabbelt
John Hauser
Adam Izraelevitz
Sagar Karandikar
Ben Keller
Donggyu Kim
John Koenig

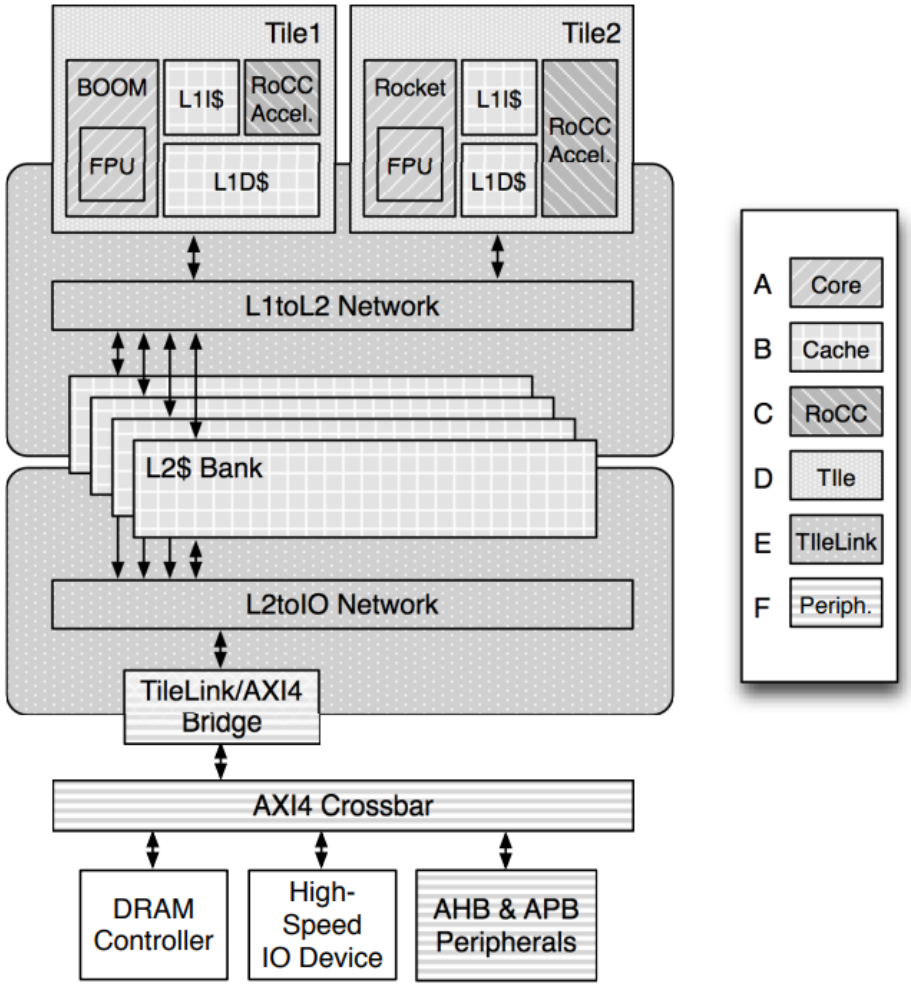
Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-17
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

April 15, 2016

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf

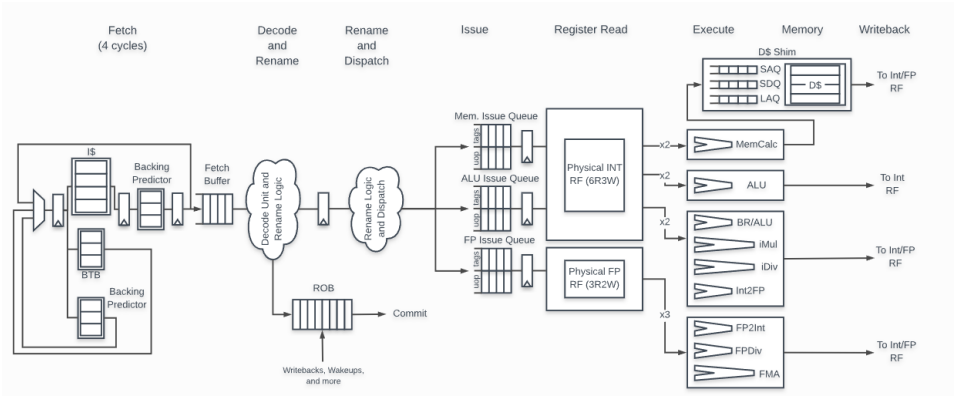
Rocket Chip架构图



1. A为Core Generator，用于生成处理器核，支持Rocket-core和BOOM两种
2. B为Cache，包括L1 Cache和L2 Cache
3. C为RoCC，即Rocket的用户自定义加速器接口，用户可以使用Chisel自行编写加速器挂载到Rocket-chip中
4. D为Tile，一个处理器核和一个L1 Cache（包括指令Cache和数据Cache）构成一个Tile，在Rocket-chip中通过复用各种Tile构建一个多核（同构或异构）的体系
5. E为TileLink，为UC Berkeley自行开发的片上总线，用于连接处理器、缓存和外设
6. F为Peripheral，包括AMBA兼容总线（AXI，AHB-Lite和APB）的发生器以及各种转换器和控制器。

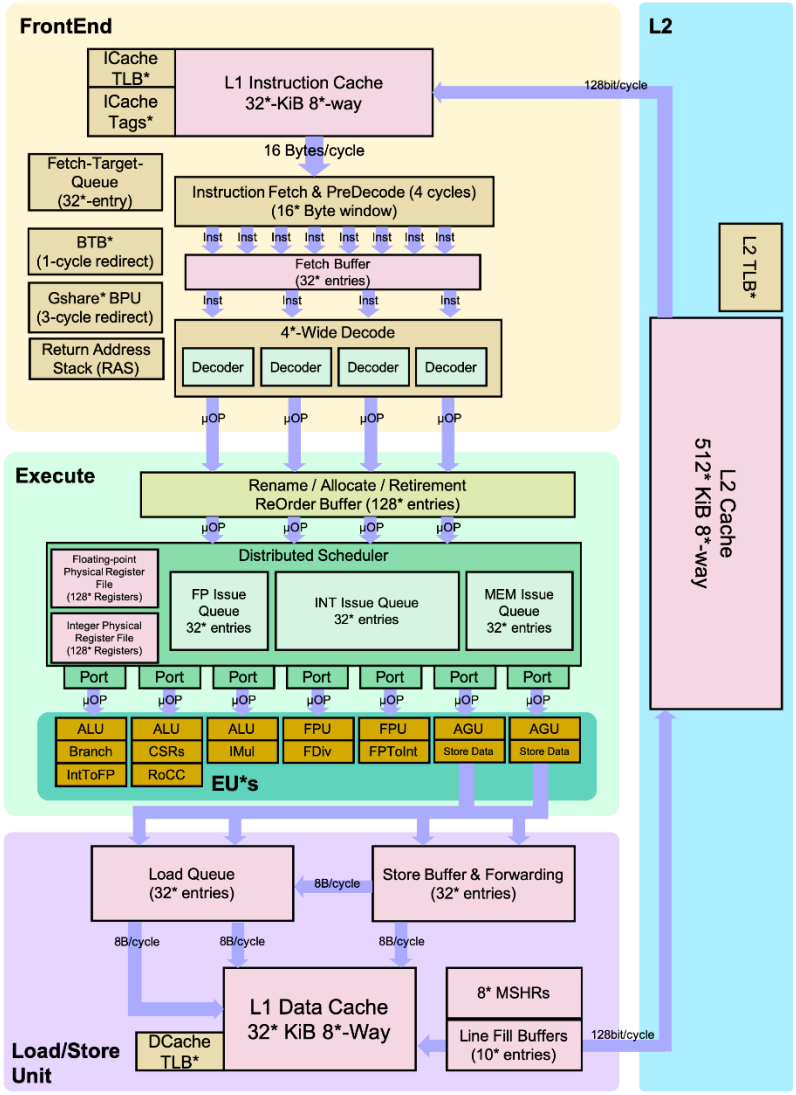
https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf

- BOOM（Berkeley Out-of-Order Machine）：超标量
  - ✓ 机器字长为64 位，支持指令集为RV64G；
  - ✓ 6 级流水线，乱序发射（10个阶段）；



Fetch、Decode/Rename/Dispatch、Issue/RegisterRead、Execute、Memory、WriteBack

https://docs.boom-core.org/en/latest/



其他RISC-V的处理器

- **Rocket**及其他RISC-V架构的产品：SiFive
- **HWACHA/**Z-Scale/V-scale/sodor/FlexPRET、YARVI: UCB提出的基于RV的类Vector处理器/单发射/教学等用途的多款开源处理器
- **LowRISC**：LowRISC是由剑桥大学为主的一些研发人员成立的非营利性组织，主要是设计发布基于RISC-V指令集的64位开源SoC
- **PULPino**：PULPino是苏黎世联邦理工大学和波罗尼亚大学联合发布的基于RISC-V的开源处理器，其处理器核RI5CY
- **RIDECORE**：东京工业大学-开源超标量RISC-V处理器（RISC-V Dynamic Execution）
- **PicoRV32**（Clifford Wolf）、**Tom Thumb** (maikmerten)
- **Hammingbird E203**：国内芯来科技开发的RISC-V MCU
- **玄铁系列 910:** 阿里平头哥
- **香山处理器**：中科院计算所

RISC-V参考资料

中国科学技术大学 University of Science and Technology of China

- Design of the RISC-V Instruction Set Architecture Waterman, Andrew Shell，UC Berkeley，UC Berkeley Electronic Theses and Dissertations Permalink

  https://escholarship.org/uc/item/7zj0b3m7

- RISC-V手册，David Patterson Andrew Waterman(中文、英文)

UC Berkeley

UC Berkeley Electronic Theses and Dissertations

Title
Design of the RISC-V Instruction Set Architecture

Permalink
https://escholarship.org/uc/item/7zj0b3m7

Author
Waterman, Andrew Shell

Publication Date
2016-01-01

Peer reviewed|Thesis/dissertation

2018

RISC-V 手册

一本开源指令集的指南

DAVID PATTERSON, ANDREW WATERMAN

翻译：勾凌睿、黄成、刘志刚
校阅：包云岗

小结

- 作业 COD RISC-V版：
  - ✓2.9，2.24，2.35, 2.40
- 思考及调研（不交）
  - ✓CPU的ISA要定义哪些内容?
  - ✓8086为什么要采用段式内存管理模式?
  - ✓Windows系统中可执行程序的格式?
  - ✓调研现在常用的处理器是大尾端还是小尾端，如x86,ARM,MIPS,PowerPC等?
  - ✓分析MIPS、X86、RISC-V指令集架构、比较相同不同点
  - ✓调研国内外典型的RISC-V处理器核、比较相同不同点
  - ✓扩展阅读Rocket及BOOM的资料
- 实验报告（交）：
  - ✓基于MIPS或者RISC-V汇编，设计一个冒泡排序程序，并用Debug工具调试执行。(MIPS仿真器/Ripes仿真)
  - ✓测量冒泡排序程序的执行时间。

Bubble sort (trace)

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	10	5	3

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	3	4	5	10

Basic idea:

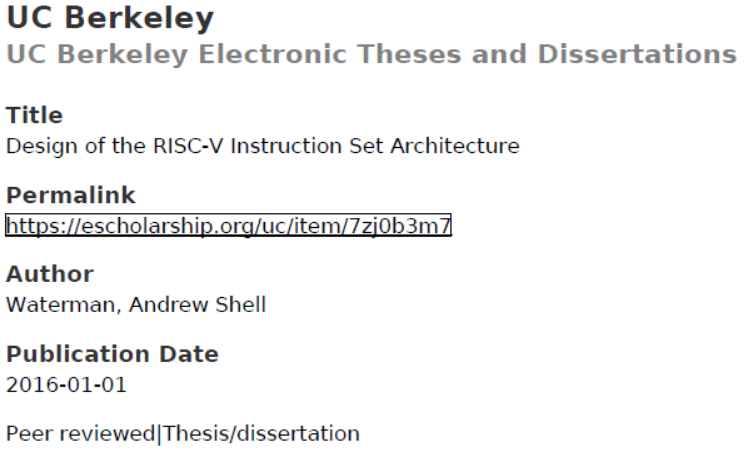
① $j \leftarrow n-1$ (index of last element in A)
② If $A[j] < A[j-1]$, swap both elements
③ $j \leftarrow j-1$, goto ② if $j > 0$
④ Goto ① if a swap occurred

① $A[0]=3$, $A[1]=4$, $A[2]=10$, $A[3]=5$, $A[4]=③$
② $A[0]=3$, $A[1]=4$, $A[2]=10$, $A[3]=③ \leftrightarrow 5$
③ $A[0]=3$, $A[1]=4$, $A[2]=10$, $A[3]=③$, $A[4]=5$
② $A[0]=3$, $A[1]=4$, $A[2]=③ \leftrightarrow 10$, $A[3]=5$, $A[4]=5$
③ $A[0]=3$, $A[1]=4$, $A[2]=③$, $A[3]=10$, $A[4]=5$
② $A[0]=3$, $A[1]=③ \leftrightarrow 4$, $A[2]=10$, $A[3]=5$, $A[4]=5$
③ $A[0]=3$, $A[1]=③$, $A[2]=4$, $A[3]=10$, $A[4]=5$
② $A[0]=③ \leftrightarrow 3$, $A[1]=4$, $A[2]=10$, $A[3]=5$, $A[4]=5$
④ Swap occured? (Yes, goto ①)
① $A[0]=3$, $A[1]=3$, $A[2]=4$, $A[3]=10$, $A[4]=⑤$

联系方式

Acknowledgements:
This slides contains materials from following lectures:
- **Computer Architecture (NUDT)**
- **CS 152 CS 252 (UC Berkeley)**

Research Area：
- 基于分布式系统, GPU，FPGA的神经网
- 人工智能和深度学习（寒武纪）芯片及



UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Design of the RISC-V Instruction Set Architecture

Permalink
https://escholarship.org/uc/item/7zj0b3m7

Author
Waterman, Andrew Shell

Publication Date
2016-01-01

Peer reviewed|Thesis/dissertation

Contact：
- 中国科学技术大学计算机学院
- 嵌入式系统实验室（西活北一楼）
- 高效能智能计算实验室 （中科大苏州研究院）

cswang@ustc.edu.cn
http://staff.ustc.edu.cn/~cswang