

《计算机组成原理》

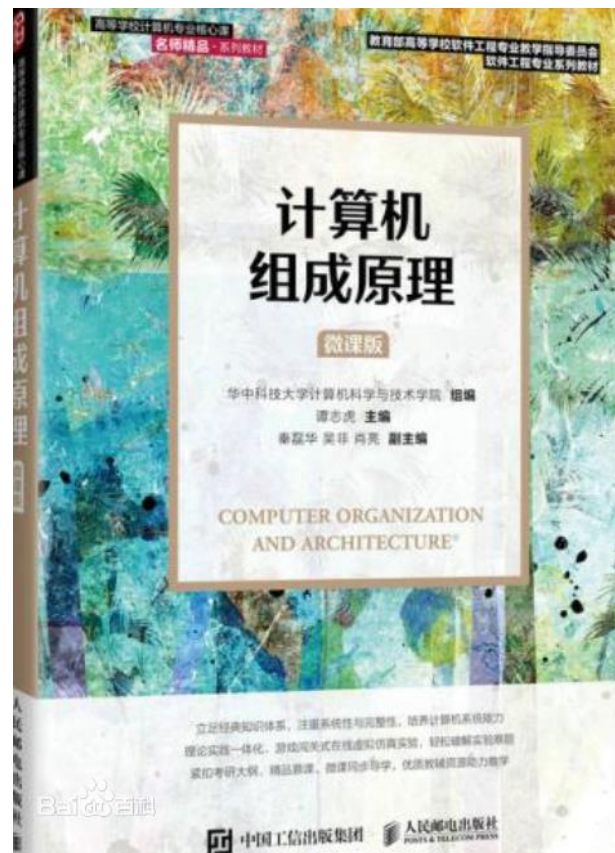
（第五讲）

厦门大学信息学院软件工程系 曾文华

2022年4月12日

目录

- 第1章 计算机系统概论
- 第2章 数据信息的表示
- 第3章 运算方法与运算器
- 第4章 存储系统
- 第5章 指令系统**
- 第6章 中央处理器
- 第7章 指令流水线
- 第8章 总线系统
- 第9章 输入输出系统



第5章 指令系统

- 5.1 指令系统概述
- 5.2 指令格式
- 5.3 寻址方式
- 5.4 指令类型
- 5.5 指令格式设计
- 5.6 CISC和RISC
- 5.7 指令系统举例

5.1 指令系统概述

- 指令

- 计算机中的指令包括：高级语言指令、汇编语言指令、机器指令、微指令；其中高级语言指令、汇编语言指令属于软件层次，机器指令、微指令属于硬件层次
- 第5章指令系统中所介绍的指令是机器指令；微指令的内容将在第6章中央处理器中介绍
- 不同级别指令之间的关系（图5.1）：

- 一条高级语言指令“编译或解释”成多条机器指令（1:m）
- 一条汇编语言指令“汇编”成一条机器指令（1:1）
- 一条机器指令由若干条微指令实现（n:1）

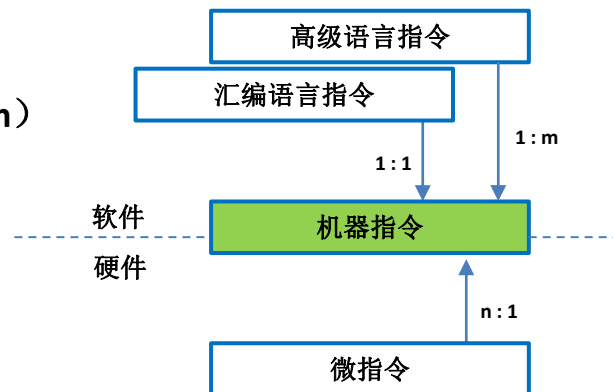


图5.1 不同级别指令之间的关系

- 指令系统

- 一台计算机中所有指令的集合称为该计算机的**指令系统**（也称**指令集**）

- 一个完善的指令系统应该满足下面的**要求**：

- ① **完备性**：要求所设计的指令系统种类齐全、功能完备，能够编写任何可计算的程序
- ② **规整性**：包括对称性和均齐性。**对称性**是指寄存器和存储单元都可被同等对待，所有指令都可以使用各种寻址方式。**均齐性**是指指令系统应提供不同数据类型的支持，方便程序设计，如算术运算指令能支持字节、字和双字整数运算，也能支持十进制数和单、双精度浮点运算
- ③ **有效性**：利用指令编写的程序能高效率地运行，方便硬件实现和编译器实现，程序占用的存储资源少，运行效率高
- ④ **兼容性**：系列计算机中新一代计算机的指令系统应该能兼容旧的指令系统（在旧计算机上开发的程序，无须修改，可以在新计算机上正确运行）
- ⑤ **可扩展性**：指令格式中的操作码要预留一定的编码空间，以便扩展指令功能

5.2 指令格式

5.2.1 指令字长度

5.2.2 指令地址码

5.2.3 指令操作码

- 指令包括操作码OP和地址码A



操作码OP: **OP**eration code

地址码A: **A**ddress

图5.2 指令的一般格式

- 操作码OP**用于解决进行何种操作的问题
- 地址码A**用于解决处理什么操作数的问题，地址码A可以包括多个操作数
- 通过何种方式获取操作数通常由**寻址方式**决定，寻址方式决定地址码A中操作数存放的位置和访问方式
- 寻址方式可以在地址码A中，如PDP-11指令集、Intel x86指令集；也可以在操作码OP中，如MIPS指令集、RISC-V指令集

• 5.2.1 指令字长度

- 指令字长度（指令长度）即指令机器码的长度，也称为指令字长
- **定长指令系统**：指令长度固定，RISC计算机多采用定长指令系统
 - 优点：结构简单，有利于CPU取指令、译码和指令顺序寻址，方便硬件执行
 - 缺点：平均指令长度较长，冗余状态较多，不容易扩展
- **变长指令系统**：指令长度可变，CISC计算机多采用变长指令系统，如Intel x86序列计算机
 - 优点：结构灵活，冗余状态较少，平均指令长度较短，可扩展性好
 - 缺点：取指令和译码不方便，取指过程可能涉及多次访存操作，下一条指令的地址必须在指令译码完成后才能确定，大大增加了硬件控制系统的设计难度
- 指令字长应该是字节的整倍数
- 根据指令字长与机器字长的关系，分为：
 - ① **半字长指令**：指令字长为机器字长的一半，CPU访问一次主存可以读取两条指令
 - ② **单字长指令**：指令字长等于机器字长，CPU访问一次主存只能读取一条指令
 - ③ **多字长指令**：指令字长为机器字长的多倍，CPU需要多个存储周期才能读取一条指令
- 通常会将最常用的指令设计为短指令格式（如半字长指令、单字长指令）

CISC: Complex Instruction Set Computer, 复杂指令系统计算机

RISC: Reduced Instruction Set Computer, 精简指令系统计算机

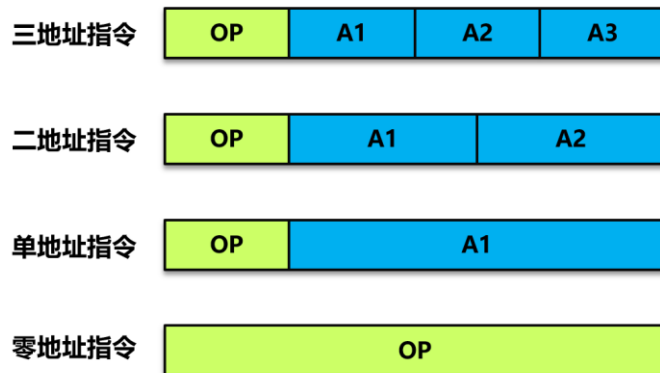
• 5.2.2 指令地址码

操作码OP

地址码A

图5.2 指令的一般格式

- 指令中地址码A的作用随指令类型和寻址方式的不同而不同
- 地址码可能是一个操作数：MOV AH, 20H
- 也可能是操作数的地址：主存地址、寄存器编号、外部设备端口的地址等
 - MOV AH, [2000H]
 - MOV AH, AL
 - IN AL, DX
- 还可能是一个用于计算地址的偏移量：MOV AX, [SI + 5]
- 根据一条指令中所含操作数地址的数量，可将指令分为：



— 1、三地址指令：

- $A_3 \leftarrow (A_1) \text{ OP } (A_2)$ 如指令： **add rd,rs,rt** ; $R[rd]=R[rs]+R[rt]$
- 三地址指令的指令长度会很长；例如，设操作码OP=6位，存储容量=16KB，则存储器地址=14位（ $2^{14}=16K$ ）；若三个地址均为存储器地址，则指令长度=6+14+14+14=48位
- 如果三地址指令中的操作数为寄存器，则可以缩短指令的长度；例如，设操作码OP=6位，寄存器数量=32个，则寄存器地址=5位（ $2^5=32$ ）；若三个地址均为寄存器地址，则指令长度=6+5+5+5=21位

— 2、双地址指令：

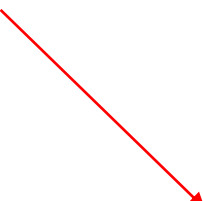
- $A_1 \leftarrow (A_1) \text{ OP } (A_2)$ 如指令： **ADD AL,BL** ; $AL \leftarrow AL+BL$
- 双地址指令有3种类型：
 - RR（寄存器-寄存器）型：MOV AL, BL
 - RS（寄存器-存储器）型：MOV [2000H], AL
 - SS（存储器-存储器）型：MOV [2000H], [3000H]
- RR型指令执行速度最快，使用最多；SS型指令执行速度最慢，使用最少

– 3、单地址指令：

- (1) 单目运算指令
 - $A_1 \leftarrow OP(A_1)$
 - 如指令： **INC AL** ; $AL \leftarrow AL + 1$
- (2) 隐含操作数的双目运算指令
 - $A_1 \leftarrow (AC) OP(A_1)$
 - 如指令： **MUL BL** ; AL 与 BL 相乘，结果送 AX （隐含操作数 AX ）

– 4、零地址指令：

- (1) 指令本身不需要操作数
 - 如指令： **NOP** ; 空操作指令
- (2) 隐含操作数的单目运算指令
 - 如指令： **DAA** ; BCD码调整指令，将 AL 中的内容进行BCD码调整（隐含 AL ）



```
MOV AL, 43H
MOV BL, 29H
ADD AL, BL      ;AL=6BH
DAA           ;AL=72H      43+29=72
```

• 5.2.3 指令操作码



图5.2 指令的一般格式

- 指令中操作码OP表示具体进行什么操作，如OP=0001表示进行加法操作，OP=0010表示进行减法操作
- 1、定长操作码**
 - 操作码长度固定，且操作码在指令中的位置也是固定的
 - 假设指令系统有m条指令，则操作码的位数n应该满足： $n \geq \log_2 m$ ；如m=128， $\log_2 m=7$ ， $n \geq 7$
- 2、变长操作码**
 - 操作码长度可变，且操作码在指令中的位置也是不固定的
 - 可以采用**扩展操作码技术**实现变长操作码，即操作码的长度随地址码数目的增加而减少；图5.3为指令字长固定的扩展操作码，指令长度=16位

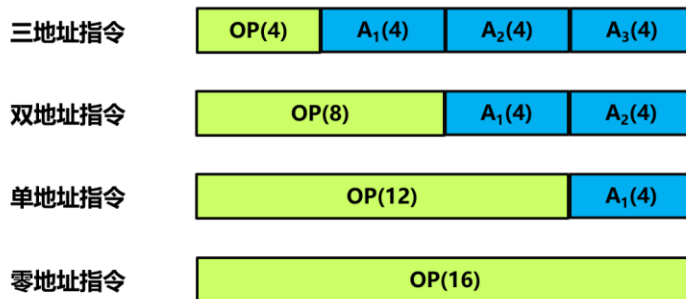


图5.3 扩展操作码

- 例5.1：假设图5.3所示的扩展操作码指令系统中有三地址指令15条、双地址指令14条、单地址指令22条，则该指令系统最多可以设计多少条零地址指令？

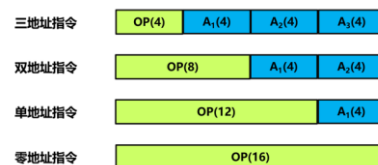


图5.3 扩展操作码

- 解：

- 双地址指令只能使用三地址指令不用的剩余状态；单地址指令只能使用双地址指令不用的剩余状态；零地址指令只能使用单地址指令不用的剩余状态
- 三地址指令的剩余状态： $2^4-15=1$
- 双地址指令的剩余状态： $1 \times 2^4-14=2$
- 单地址指令的剩余状态： $2 \times 2^4-22=10$
- 零地址指令的数目： $10 \times 2^4=160$
- 因此该指令系统最多可以设计160条零地址指令

160条

```
1111 1111 0110 0000
1111 1111 0110 0001
.....
1111 1111 0110 1111
```

```
1111 1111 0111 0000
1111 1111 0111 0001
.....
1111 1111 0111 1111
```

15条

```
0000 XXXX XXXX XXXX
0001 XXXX XXXX XXXX
0010 XXXX XXXX XXXX
0011 XXXX XXXX XXXX
0100 XXXX XXXX XXXX
0101 XXXX XXXX XXXX
0110 XXXX XXXX XXXX
0111 XXXX XXXX XXXX
1000 XXXX XXXX XXXX
1001 XXXX XXXX XXXX
1010 XXXX XXXX XXXX
1011 XXXX XXXX XXXX
1100 XXXX XXXX XXXX
1101 XXXX XXXX XXXX
1110 XXXX XXXX XXXX
```

三地址指令

14条

```
1111 0000 XXXX XXXX
1111 0001 XXXX XXXX
1111 0010 XXXX XXXX
1111 0011 XXXX XXXX
1111 0100 XXXX XXXX
1111 0101 XXXX XXXX
1111 0110 XXXX XXXX
1111 0111 XXXX XXXX
1111 1000 XXXX XXXX
1111 1001 XXXX XXXX
1111 1010 XXXX XXXX
1111 1011 XXXX XXXX
1111 1100 XXXX XXXX
1111 1101 XXXX XXXX
```

双地址指令

22条

```
1111 1110 0000 XXXX
1111 1110 0001 XXXX
1111 1110 0010 XXXX
1111 1110 0011 XXXX
1111 1110 0100 XXXX
1111 1110 0101 XXXX
1111 1110 0110 XXXX
1111 1110 0111 XXXX
1111 1110 1000 XXXX
1111 1110 1001 XXXX
1111 1110 1010 XXXX
1111 1110 1011 XXXX
1111 1110 1100 XXXX
1111 1110 1101 XXXX
1111 1110 1110 XXXX
1111 1110 1111 XXXX
```

单地址指令

单地址指令

零地址指令

```
1111 1111 1111 0000
1111 1111 1111 0001
.....
1111 1111 1111 1111
```

5.3 寻址方式

5.3.1 指令寻址方式
5.3.2 操作数寻址方式

- 指令和数据（称为操作数）是存放在主存中的，只有获得指令和操作数在主存的地址（称为**有效地址EA**，Effective Address），CPU才能访问所需的指令和数据
- **寻址方式**就是寻找指令或操作数有效地址的方法
- 寻址方式分为**指令寻址方式**和**操作数寻址方式**

• 5.3.1 指令寻址方式

– 1、顺序寻址方式

- 大部分程序中的机器指令序列在主存中是按顺序存放的；因此只要知道第一条指令的有效地址，通过增加一条指令所占用主存单元的数量，就可以知道下一条指令的有效地址
- 这种计算有效地址的方法称为**顺序寻址方式**（图5.4a）
- **程序计数器**：PC，Program Counter，用于保存指令地址（主存地址）
- 假设主存按照字节进行编址；如果是单字节指令，则PC+1就是下一条指令的地址；如果是双字节指令，则PC+2就是下一条指令的地址；以此类推
- 因此，PC应具有自动加1的功能

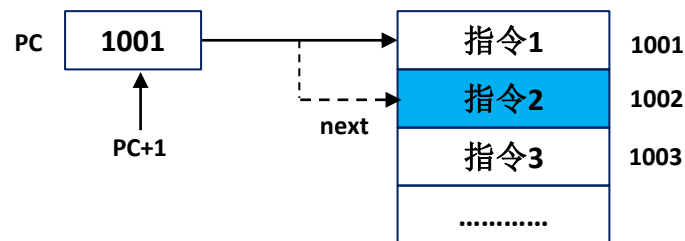


图5.4a 顺序寻址

– 2、跳跃寻址方式

- 如果程序中有分支或转移，就会改变程序的执行顺序，此时就需要采取**跳跃寻址方式**（图5.4b）
- 跳跃寻址方式时，下一条指令的地址由指令本身及指令需要测试的条件决定
- 无条件转移指令、条件转移指令都采用跳跃寻址方式
- 在图5.4b中，取出“**jmp 1003**”指令时，PC先是变为1002（PC具有自动加1的功能）；之后根据“**jmp 1003**”指令，又使PC变为1003；执行完“**jmp 1003**”指令后，不是执行指令2，而是跳转到指令3执行

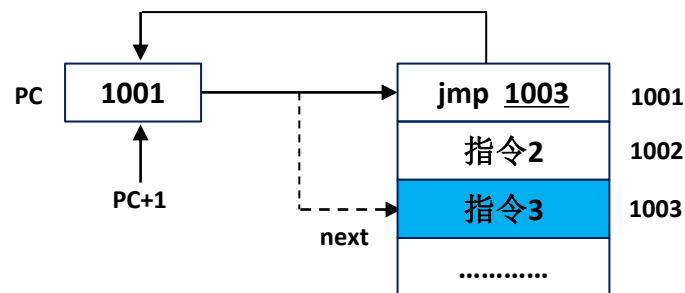


图5.4b 跳跃寻址

• 5.3.2 操作数寻址方式

- 操作数寻址方式就是形成操作数**有效地址**的方法
- 操作数的来源通常有**3**种情况：
 - ① 操作数直接来自指令地址字段：MOV AL, **23H**
 - ② 操作数存放在寄存器中：MOV AL, **BL**
 - ③ 操作数存放在存储器中：MOV AL, **[2300H]**
- 常用的操作数寻址方式有：立即寻址、直接寻址、寄存器寻址、间接寻址、寄存器间接寻址、相对寻址、变址寻址、基址寻址、堆栈寻址等
- 此时，可将指令的地址码**A**字段细分为**寻址方式I**字段和**形式地址D**字段（图5.5）；寻址方式I字段也称为寻址方式特征码
- 假设最终的操作数为**S**，有效地址为**EA**，则有**S=(EA)**；这里**()**表示访问地址为**EA**的主存或寄存器的内容



图5.2 指令的一般格式

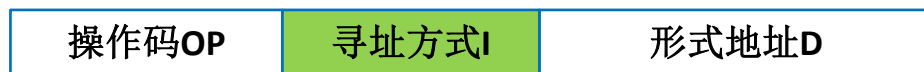


图5.5 含寻址方式字段的单地址指令格式

– 1、立即寻址

- 例如： **MOV EAX, 2008H** ； 传送类指令，其中操作数**2008H**采用立即寻址方式
- 图5.6a为立即寻址方式的指令格式，假设寻址方式特征码**I=000**
- 立即寻址方式时，形式地址**D**就是操作数**S**本省，即**S=D=2008H**；执行指令“**MOV EAX, 2008H**”后，**EAX=2008H**
- 立即寻址方式指令执行阶段不需要访问主存



图5.6a 立即寻址

– 2、直接寻址

- 例如：MOV EAX, **[2008H]** ； 传送类指令，其中操作数**[2008H]**采用直接寻址方式
- 图5.6b为直接寻址方式的指令格式，假设寻址方式特征码=001
- 直接寻址方式时，操作数**S**在主存中，操作数地址（有效地址EA）由形式地址**D**直接给出，即有效地址**EA=D=2008H**，**S=[2008H]**
- 假设主存2008H的内容为A0A0H，执行指令“MOV EAX, [2008H]”后，EAX=A0A0H
- 直接寻址方式的特点是地址直观，指令执行阶段需要访问一次主存

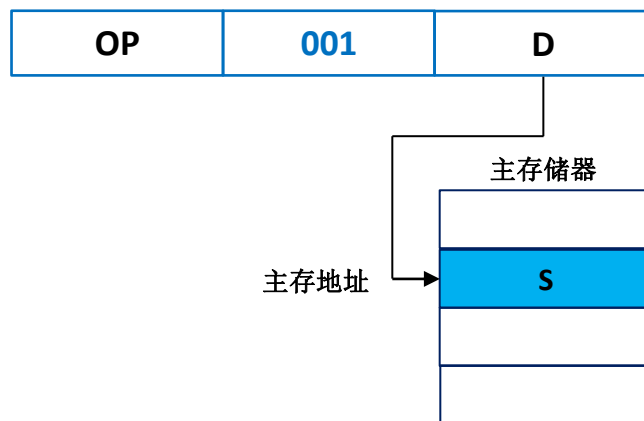


图5.6b 直接寻址

– 3、寄存器寻址

- 例如：MOV EAX, ECX；传送类指令，其中操作数EAX、ECX采用寄存器寻址方式
- 图5.6c为寄存器寻址方式的指令格式，假设寻址方式特征码=010
- 寄存器寻址方式时，形式地址D表示寄存器编号，操作数S为寄存器中的内容，即有效地址EA=D，S=R[D]
- 寄存器寻址方式指令执行阶段不需要访问主存

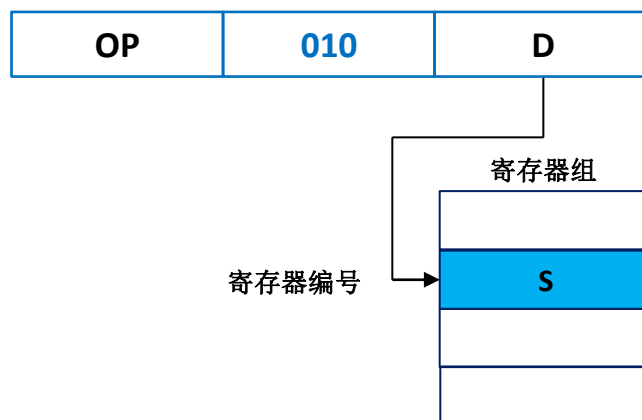


图5.6c 寄存器寻址

– 4、间接寻址

- 例如：MOV EAX, @2008H ； 传送类指令，其中操作数@2008H采用间接寻址方式
- 图5.7为间接寻址方式的指令格式，假设寻址方式特征码=011
- 间接寻址方式时，形式地址D给出的不是操作数的有效地址，而是操作数的间接地址；即有效地址EA=(D)，操作数S=(EA)
- 例5.2：假设主存地址2008H中的内容为A0A0F000H（32位），而主存地址A0A0F000H中的内容为5000H，则操作数S=5000H；即执行指令“MOV EAX, @2008H”后，EAX=5000H
- 间接寻址方式指令执行阶段需要访问两次主存

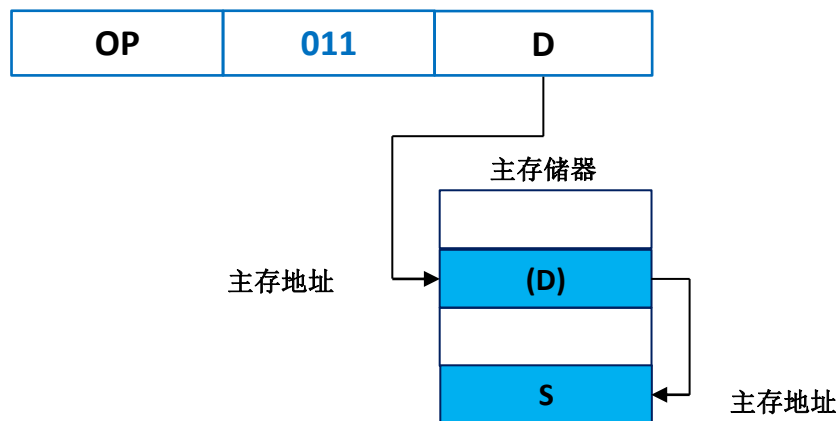


图5.7 间接寻址

– 5、寄存器间接寻址

- 例如：MOV AL, [EBX] ； 传送类指令，其中操作数[EBX]采用寄存器间接寻址方式
- 图5.8为寄存器间接寻址方式的指令格式，假设寻址方式特征码=100
- 寄存器间接寻址方式时，形式地址D给出的是寄存器编号，有效地址为寄存器的内容，即有效地址 $EA=R[D]$ ，操作数 $S=(EA)$
- 假设EBX=2010H，主存2010H的内容为60H，则操作数 $S=60H$ ；即执行指令“MOV AL, [EBX]”后， $AL=60H$
- 寄存器间接寻址方式指令执行阶段只需要访问一次主存

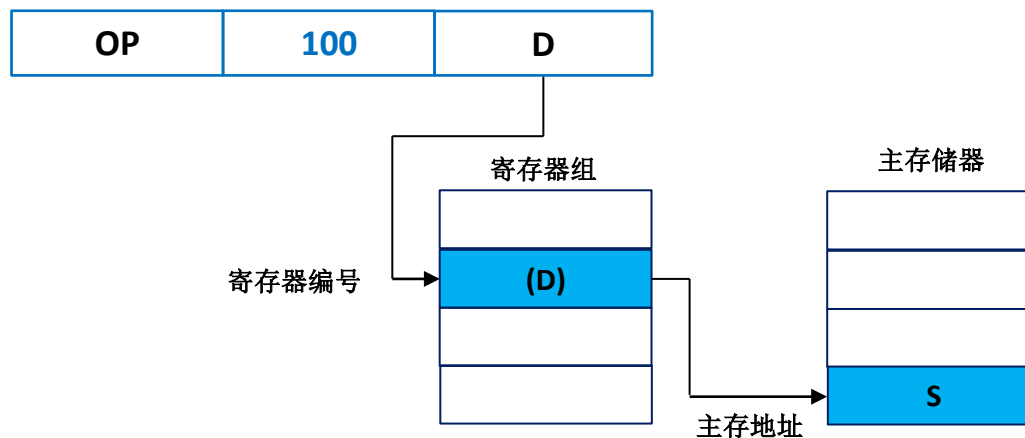


图5.8 寄存器间接寻址

– 6、相对寻址

- 例如：MOV EAX, 32[PC] ； 传送类指令，其中操作数32[PC] 采用相对寻址方式
- 图5.9为相对寻址方式的指令格式，假设寻址方式特征码=101
- 相对寻址方式时，形式地址D给出的是偏移量，有效地址为程序计数器PC的内容加上偏移量，即有效地址EA=PC+D；注意：这里程序计数器PC的内容为取指令完成后PC的值
- 假设指令“MOV EAX, 32[PC]”在取指令完成后，PC的内容为2000H，D=32H，EA=PC+D=2000H+32H=2032H，主存2032H的内容为5000H；则执行指令“MOV EAX, 32[PC]”后，EAX=5000H
- 相对寻址经常用于跳转指令，如指令“JMP LOOP”，操作数LOOP就是采用相对寻址方式

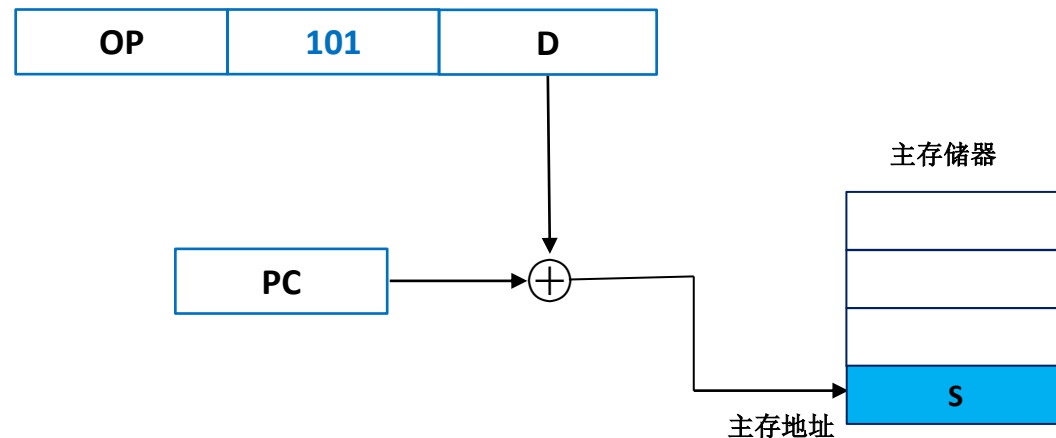


图5.9 相对寻址

- 例5.3: 某计算机指令字长为定长16位, 内存按字节寻址, 指令中的数据采用补码表示, 且PC的值在取指令阶段完成修改。请完成下列有关相对寻址的问题:
- (1) 若采用相对寻址指令的当前地址为2003H, 且要求数据有效地址为200AH, 则该相对寻址指令的形式地址字段的值为多少?
- (2) 若采用相对寻址转移指令的当前地址为2008H, 且要求转移后的目标地址为2001H, 则该相对寻址指令的形式地址字段的值为多少?

• 解:

• (1)

– 有效地址EA=程序计数器PC+形式地址D, 因此 $D=EA-PC$

– 采用相对寻址指令的当前地址为2003H, 因为指令字长为定长16位, 内存按字节寻址; 因此, 取指令后, PC的值= $2003H+16\text{位}=2003H+2H=2005H$

– 所以, $D = EA-PC = 200AH-2005H = 5H$; 该相对寻址指令的形式地址字段的值为5H

• (2)

– 当前地址为2008H, 取指令后, PC的值= $2008H+16\text{位}=2008H+2H=200AH$

– 所以, $D = EA-PC = 2001H-200AH = -9 = F7H$; 该相对寻址指令的形式地址字段的值为F7H

– 7、变址寻址

- 例如：MOV EAX, 32[ESI] ； 传送类指令，其中操作数32[ESI]采用变址寻址方式
- 图5.10为变址寻址方式的指令格式，假设寻址方式特征码=110，地址码字段增加一个变址寄存器X编号字段
- 变址寻址方式时，有效地址为变址寄存器X的内容加上形式地址D，即有效地址 $EA=R[X]+D$ ；指令“MOV EAX, 32[ESI]”中，ESI为变址寄存器，32为形式地址；假设ESI的内容为2000H， $EA=2000H+32H=2032H$ ；假设主存2032H的内容为5000H；则执行指令“MOV EAX, 32[ESI]”后，EAX=5000H
- 变址寻址方式中，D通常是不变的，X的内容变化
- 变址寻址是面向用户的，主要用于解决程序循环问题；变址寄存器的内容由用户设置，程序执行过程中，用户通过改变变址寄存器的内容，实现指令或操作数的寻址

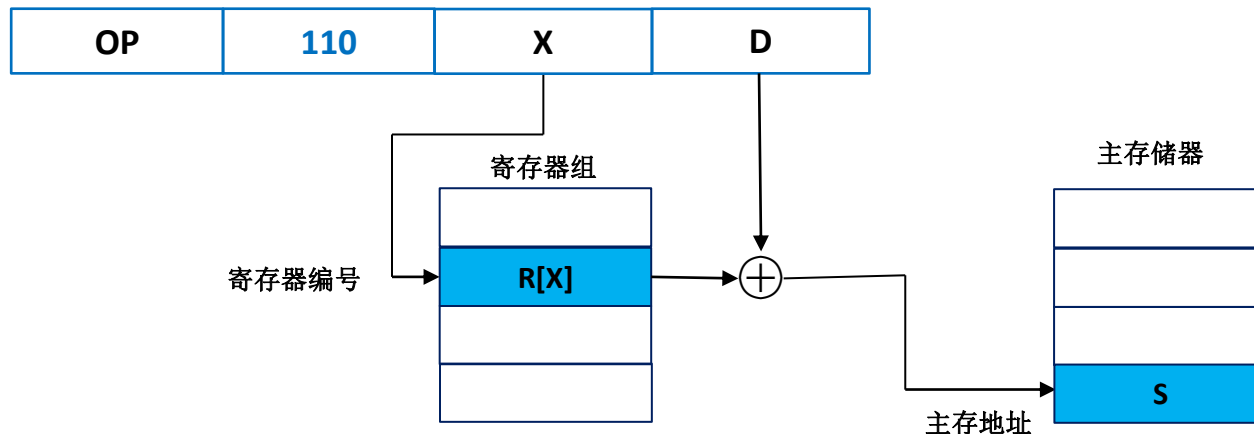


图5.10a 变址寻址

– 8、基址寻址

- 例如：MOV EAX, 32[EBX] ； 传送类指令，其中操作数32[EBX]采用基址寻址方式
- 基址寻址方式的指令格式和变址寻址方式相同，寻址过程也与变址寻址方式相同，此时地址码字段增加一个基址寄存器B编号字段，如图5.10b所示
- 基址寻址方式中，D通常是变化的，B的内容不变（而变址寻址方式中，D通常是不变的，X的内容变化）
- 基址寻址是面向系统的，主要用于程序的重定位；如第四章的页表基址寄存器PTBR，页表项地址PTEA=PTBR+虚页号VPN，其中PTBR是固定的，VPN是变化的（图4.48）
- 相对寻址、变址寻址、基址寻址等3种寻址方式非常类似，也称为偏移寻址

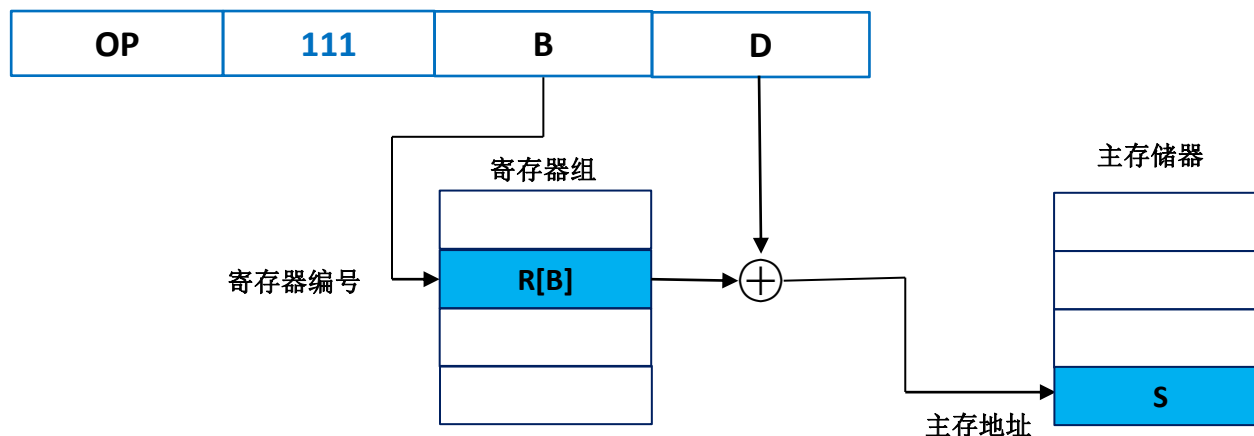
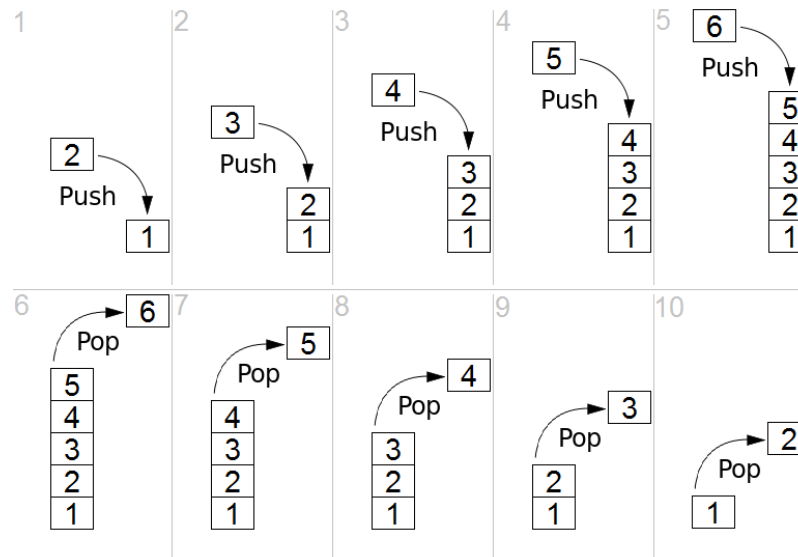
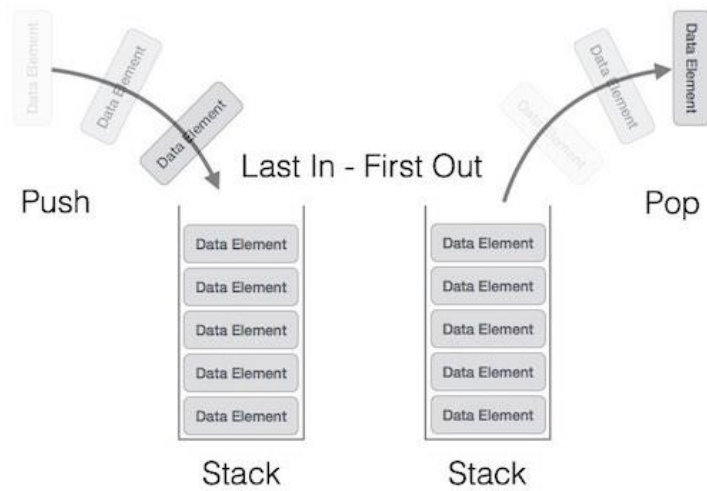


图5.10b 基址寻址

– 9、堆栈寻址

- 堆栈（Stack）以先进后出（First In Last Out）的方式存储数据，堆栈通常有存储器堆栈和寄存器堆栈两种



- (1) 存储器堆栈

- 图5.11为存储器堆栈**进出栈**的操作过程，其中SP为堆栈指针寄存器（Stack Pointer）
- 入栈操作： $SP=SP-1$ ， $M[SP]=R$ ； 假设入栈数据为1个字节（8位）
- 出栈操作： $M[SP]=R$ ， $SP=SP+1$ ； 假设出栈数据为1个字节（8位）
- 注意：如果进出栈的数据为16位，则为SP-2（SP+2）；如果进出栈的数据为32位，则为SP-4（SP+4）

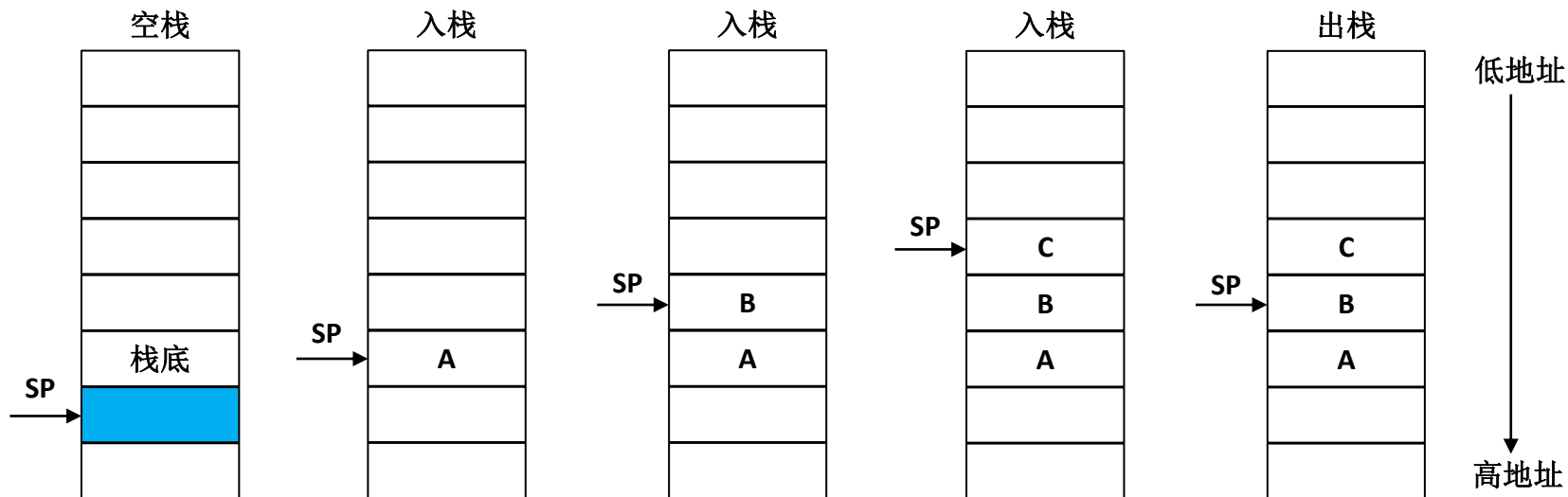


图5.11 存储器堆栈进出栈的操作过程

• (2) 寄存器堆栈

– 图5.12为寄存器堆栈**进出栈**的操作过程，寄存器堆栈不需要堆栈指示器SP

– 寄存器堆栈与存储器堆栈的不同点：

① 寄存器堆栈栈顶固定不动，而存储器堆栈栈顶随着堆栈操作而移动

② 进行堆栈操作时，寄存器堆栈中的数据移动，而存储器堆栈中的数据不动

③ 寄存器堆栈速度快，但容量有限；存储器堆栈速度慢，但容量很大

④ 寄存器堆栈必须采用专用堆栈指令进行控制，存储器堆栈则不一定

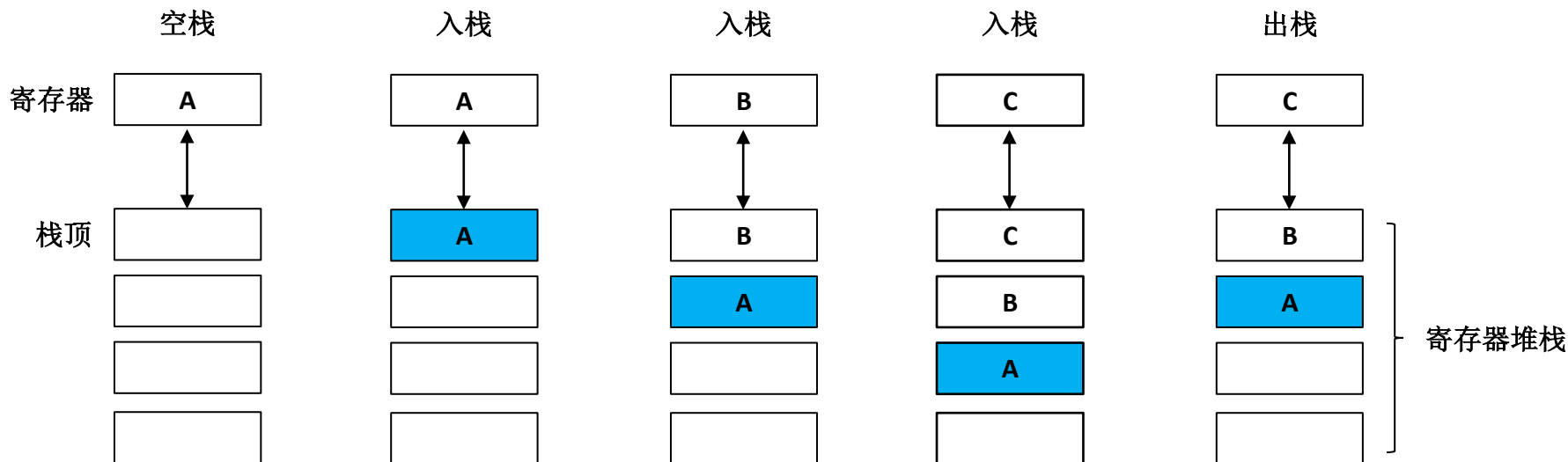


图5.12 寄存器堆栈进出栈的操作过程

– 10、其他寻址

- (1) 变址 + 间接寻址方式

变址寻址: $R[X]+D$

间接寻址: $(R[X]+D)$

- 先进行变址寻址，再进行间接寻址；有效地址 $EA=(R[X]+D)$ ，其中 $R[X]$ 为变址寄存器 X 的内容， D 为形式地址

- (2) 间接 + 变址寻址方式

间接寻址: (D)

变址寻址: $R[X]+(D)$

- 先进行间接寻址，再进行变址寻址；有效地址 $EA=R[X]+(D)$ ，其中 $R[X]$ 为变址寄存器 X 的内容， D 为形式地址

- (2) 相对 + 间接寻址方式

相对寻址: $PC+D$

间接寻址: $(PC+D)$

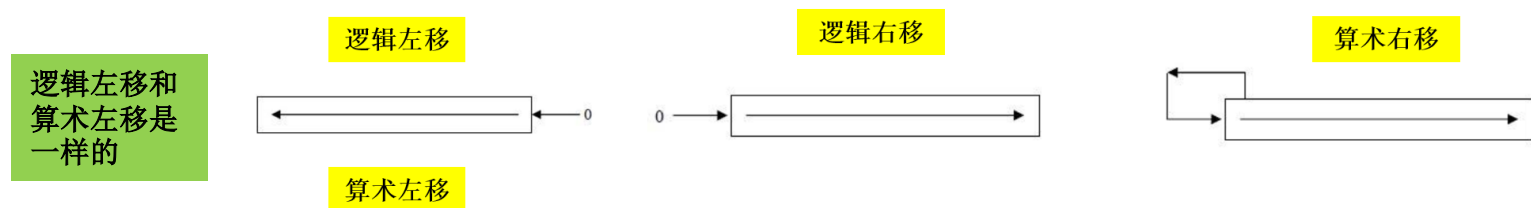
- 先进行相对寻址，再进行间接寻址；有效地址 $EA=(PC+D)$ ，其中 PC 为程序计数器， D 为形式地址

5.4 指令类型

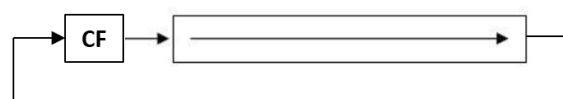
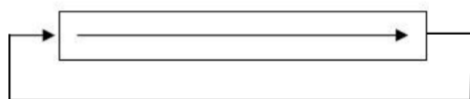
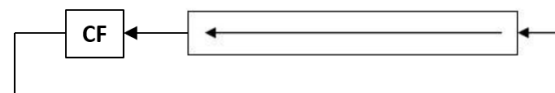
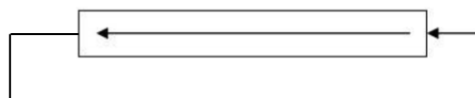
- **指令系统**决定了计算机的基本功能，不同类型的计算机，其指令系统之间的差异很大
- 有的计算机指令类型多，功能丰富，包含几百条指令；有的计算机指令类型少，功能简单，只有几十条指令
- 一个完善的**指令系统**应包括如下的**指令类型**：
 - **1、算术逻辑运算指令**
 - 包括定点、浮点数的加、减、乘、除等算术运算指令，与、或、非、异或等逻辑运算指令，有些计算机还专门设置了十进制运算指令
 - 例如： **ADD AL,BL SUB AL,BL AND AL,BL XOR AL,BL**

• 2、移位操作指令

- 算术移位指令是针对带符号数的移位操作，包括**算术左移**指令和**算术右移**指令
- 逻辑移位指令是针对无符号数的移位操作，包括**逻辑左移**指令和**逻辑右移**指令



- 循环指令（小循环），包括**循环左移**和**循环右移**
- 带进位循环指令（大循环），包括**带进位循环左移**和**带进位循环右移**



- ### 3、数据传送指令

- 主要完成两个部件之间的数据传送，如寄存器与寄存器之间、寄存器与存储器之间、存储器与存储器之间
- 通用数据传送指令（**MOV**）：支持寄存器与寄存器之间、寄存器与存储器之间、存储器与存储器之间的数据传送
- **LOAD**（存储器读）/**STORE**（存储器写）指令：用于访问存储器，此时**MOV**指令只能用于寄存器之间的数据传送
- 交换指令（**SWAP**）：可以实现双向的数据传送
- 串传送指令（**MOVS**）：通过加上重复前缀**REP**（**REP MOVS**），可以实现将存储器中的数据块从一个区域传送到另一个区域
- 有的计算机将堆栈指令（**PUSH**、**POP**）、寄存器/存储单元清零指令（**CLR**），也归属于数据传送指令

- 4、堆栈操作指令

- 压栈指令（**PUSH**）：如**PUSH AX**，将寄存器**AX**的内容压入到堆栈中
- 出栈指令（**POP**）：如**POP AX**，将堆栈中的内容弹出到寄存器**AX**中

- 5、字符串处理指令

- 字符串处理指令属于非数值处理指令
- 包括字符传送、字符串比较、字符串查找、字符串抽取、字符串转换等指令
- 不是所有的计算机都支持字符串处理指令

• 6、程序控制指令

– (1) 转移指令

- 包括：**无条件转移指令**（例如，**JMP LOOP**）和**条件转移指令**（例如，**JZ LOOP**；结果为零，则转移）

– (2) 循环控制指令

- 例如，**LOOP L1**，该指令每执行一次，循环计数器**ECX**减1，然后判断**ECX**是否为0；若不为0，则转到**L1**处继续执行；若为0，则结束循环，执行**LOOP L1**指令的下一条指令

```
L1: MOV AL, BL
.....
.....
LOOP L1
SUB CL, DL
```

– (3) 子程序调用和返回指令

- 包括**CALL指令**（子程序调用指令，也称转子指令，或过程调用指令）和**RET指令**（子程序返回指令）
- **CALL**指令的下一条指令的地址称为**断点**；为了保证**RET**指令能够正确返回到断点，**CALL**指令应具有保护断点的功能；通常**CALL**指令会将断点压入堆栈，**RET**指令则从堆栈中取出断点，从而实现正确返回断点
- **CALL**指令（转子指令）与转移指令的**区别**：
 - ① 转移的位置不同，转移指令在同一个程序内转移，**CALL**指令在不同程序之间转移
 - ② 转移指令不需要返回原处，而**CALL**指令需要返回原处，因此**CALL**指令需要保护断点
 - ③ **CALL**指令和**RET**指令通常是无条件的，而条件转移指令是有条件的

- 7、输入输出指令

- 输入指令：如IN AL, DX；将以寄存器DX为端口地址的输入设备的内容送到寄存器AL中
- 输出指令：如OUT DX, AL；将寄存器AL的内容送到以寄存器DX为端口地址的输出设备中
- 当外部设备与主存采用统一编址时，则不需要设置专用的输入输出指令（IN和OUT），而是可以使用访存指令实现输入和输出指令的功能

- 8、其他指令

- 停机、等待、空操作、开中断、关中断、自陷、置条件码、特权指令等
- 特权指令一般只能用于操作系统或其他系统软件，而不直接提供给用户使用

5.5 指令格式设计

- **指令系统**是程序员所能看到的计算机的主要属性，它在很大程度上决定了整个计算机系统具有的基本功能和程序性能
- 设计指令系统要**考虑**:
 - ① 指令的完备性、规整性、有效性、兼容性和可扩展性
 - ② 支持哪些指令（**5.4** 指令类型）
 - ③ 支持哪些数据类型和寻址方式（**5.3.2** 操作数寻址方式）
 - ④ 合理的指令格式（**5.2** 指令格式）
 - ⑤ 不仅要方便程序员进行程序设计，也要有利于编译系统的设计、有利于简化硬件实现、有利于节省程序存储空间
- 指令一般由**操作码**和**地址码**组成，指令格式设计包括：指令编码格式的设计、操作码的设计、地址码的设计、寻址方式的设计

- 1、指令编码格式的设计

- (1) 定长指令格式

- 指令长度固定：如MIPS 32指令系统，所有的指令对应的机器码都是32位
 - 优点：有利于简化硬件，尤其有利于简化指令译码部件的设计
 - 缺点：指令系统的平均长度长，容易出现冗余码点，不利于扩展

- (2) 变长指令格式

- 指令长度可变：如Intel x86指令系统，最短的指令为单字节，最长的指令为15个字节
 - 优点：指令平均长度短，指令码点冗余少，易于扩展
 - 缺点：指令格式不规整，不同指令的取指令时间可能不同，控制方式较为复杂

- (3) 混合编码指令格式

- 是定长指令格式和变长指令格式的综合
 - 提供若干长度固定的指令字
 - 既能减少目标代码的长度（变长指令格式），又能降低译码复杂度（定长指令格式）

- 2、操作码的设计

- 满足完备性是操作码设计的基本要求（**完备性**：要求所设计的指令系统种类齐全、功能完备，能够编写任何可计算的程序）
- **定长结构**的操作码：操作码长度固定
- **变长结构**的操作码：操作码长度可变

- 3、地址码的设计

- 地址码（操作数）的数量：三地址、双地址、单地址、零地址
- 地址码设计的目的：利用有限的位宽提供**更大的寻址范围**

- 4、寻址方式的设计

- 一种是将寻址方式与操作码一起编码
- 另一种是设置专门的寻址方式字段（图5.5）



图5.5 含寻址方式字段的单地址指令格式

- 例5.4：某计算机字长为16位，内存为64KB，指令采用单字长、单地址结构，要求至少能支持80条指令和直接、间接、相对、变址等4种寻址方式。请设计指令格式并计算每种寻址方式能访问的主存空间范围。

- 解：

- 该指令系统为定长指令格式，指令长度=单字长=16位
- 因为至少能支持80条指令，因此操作码OP字段为7位 ($2^7 > 80$; $2^6 < 80$)
- 因为要支持4种寻址方式，因此寻址方式I字段为2位 ($2^2 = 4$)
- 因为是单地址结构，指令的格式如图5.13所示，其中形式地址D=16-7-2=7位
- 每种寻址方式的访存范围：
 - ① 直接寻址的访存空间范围：有效地址EA=D=7位，0~127
 - ② 间接寻址的访存空间范围：有效地址EA=(D)=16位，0~65535（存储器的宽度=字长=16位）
 - ③ 相对寻址的访存空间范围：有效地址EA=PC+D=16位，0~65535（程序计数器PC的长度为16位）
 - ④ 变址寻址的访存空间范围：有效地址EA=R[X]+D=16位，0~65535（变址寄存器X的长度为16位）

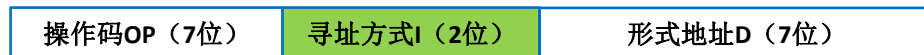


图5.13 指令格式

5.6 CISC和RISC

5.6.1	CISC
5.6.2	RISC

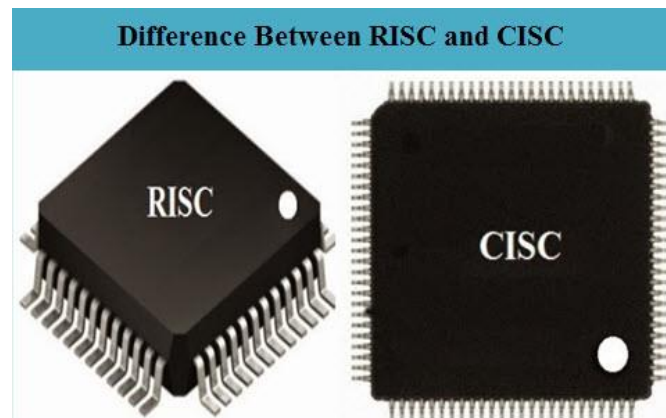
• 5.6.1 复杂指令系统计算机（CISC）

- CISC: Complex Instruction Set Computer, 复杂指令系统计算机
- 计算机系统设计者在设计指令系统时, 增加了越来越多的功能强大的**复杂指令**, 以及更多的寻址方式, 以满足来自不同方面的**需求**:
 - ① 更好地支持高级语言, 增加语义接近高级语言的指令
 - ② 简化编译, 增加语义接近高级语言的指令
 - ③ 满足系列计算机软件向后兼容（向上兼容）的需求, 新的计算机指令只能增加不能减少, 指令的数量越来越多
 - ④ 对操作系统的支持, 要求指令系统提供相应功能的指令, 如多媒体指令、3D指令等
 - ⑤ 为在有限指令长度内基于扩展法实现更多指令, 只有最大限度地压缩地址码长度, 从而设计多种寻址方式

- 因此，计算机的指令系统越来越庞大、复杂，称为复杂指令系统计算机（CISC）
- 复杂指令系统计算机（CISC）具有如下特点：
 - ① 指令系统复杂庞大，指令数目多达200-300条
 - ② 寻址方式多
 - ③ 指令格式多
 - ④ 指令字长不固定
 - ⑤ 对访存指令不加限制
 - ⑥ 各种指令使用频率相差大
 - ⑦ 各种指令执行时间相差大
 - ⑧ 大多数采用微程序控制器

• 5.6.2 精简指令系统计算机（RISC）

- RISC: Reduced Instruction Set Computer, 精简指令系统计算机
- “80-20” 规律: 80%的程序只用到了20%的指令
- 精简指令系统计算机（RISC）体系结构的**基本思想**: 针对CISC指令系统指令种类太多、指令格式不规范、寻址方式太多的缺点, 通过减少指令种类、规范指令格式和简化寻址方式, 来方便处理器内部的并行处理, 从而大幅度提高处理器的性能



— 精简指令系统计算机（RISC）的特点：

- ① 优先选择使用频率最高的一些简单指令，以及一些很有用但不复杂的指令，避免使用复杂指令
- ② 大多数指令在一个时钟周期内执行完成
- ③ 采用LOAD/STORE结构，只允许LOAD/STORE指令访问主存，其余指令只能对寄存器操作数进行处理
- ④ 采用简单的指令格式和寻址方式，指令长度固定
- ⑤ 固定的指令格式；指令长度、格式固定，可简化指令的译码逻辑，有利于提高流水线的执行效率；为了便于编译的优化，常采用三地址指令格式
- ⑥ 面向寄存器的结构；为减少访问主存，CPU内应设大量的通用寄存器
- ⑦ 采用硬布线控制逻辑；由于指令系统的精简，控制部件可由组合逻辑实现，不用或少用微程序控制，这样可使控制部件的速度大大提高
- ⑧ 注重编译的优化，力求有效地支持高级语言程序

5.7 指令系统举例

- 5.7.1 PDP-11指令系统
- 5.7.2 Intel x86指令系统
- 5.7.3 MIPS指令系统
- 5.7.4 RISC-V指令系统

• 5.7.1 PDP-11指令系统

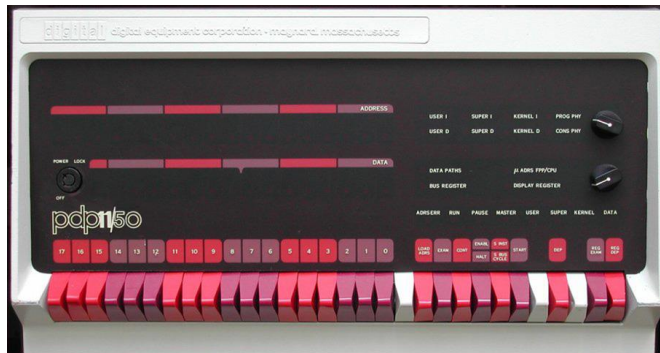
- PDP-11是DEC公司在1970年推出的一款经典的16位小型计算机，是IBM PC出现之前最为流行的小型计算机



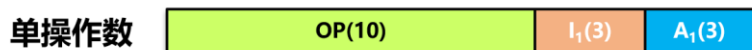
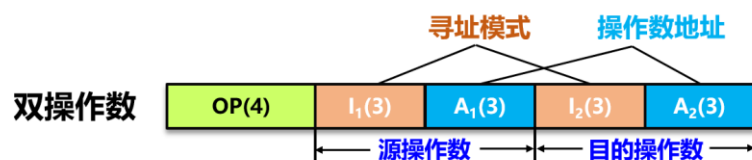
DEC公司



- 美国**DEC公司**（Digital Equipment Corporation，**数字设备公司**）。1998年1月DEC公司被**康柏**以96亿美元的价格收购，2001年**惠普**康柏宣布合并。
- DEC公司于1957年诞生，1958年初，即运出了第一批产品-数字实验室和数字系统组件，第一年的销售额为9.4万美元，公司获得了一点盈利，尽管很少。1962年，是DEC公司的第五个财政年度，公司的销售额是650万美元，净利润为80万美元；1963年，DEC创利120万美元；1964年销售额上升为1100万美元，但利润下降为90万美元。DEC公司处于发展中，资金紧张，支出大于收入，1963年6月，DEC公司需要一笔现金，向ARD公司贷款30万美元，分三年还清。
- 1966年，安德森决定退出DEC公司。1965年，DEC公司的销售收入为1500万美元，1966年增长到2300万美元。
- 1966年8月16日，DEC公司公开招股上市，价格为每股22美元，ARD公司拥有175万股，占公司股份总额的65%，价值3850万美元，增值达550倍，奥尔森拥有35万股，占13%，价值770万美元；安德森拥有14万股，上市后即可转让，价值300万美元。



- PDP-11指令系统属于典型的**16位CISC指令系统**，指令字长通常为单字长（16位），有单操作数（单地址）、双操作数（双地址）两种指令格式，如图5.14a
- 通过采用变址寻址方式，可以增加1~2个立即数index字段，从而将指令扩展为双字长（32位）、三字长（48位）指令格式，如图5.14b



(a) 单字长指令格式



(b) 双字长、三字长指令格式

图5.14 PDP-11指令格式

- 寻址特征字段I（寻址模式）为3位，因此有**8种寻址方式**，如表5.1所示
- 寄存器字段A（操作数地址）为3位，可寻址**8个寄存器** $R_0 \sim R_7$ ；其中 $R_0 \sim R_5$ 为通用寄存器， R_6 为SP（堆栈指针寄存器）， R_7 为PC（程序计数器）
- PDP-11的**自增、自减**寻址方式，方便对堆栈进行操作，也影响了C语言的语法（例如： $i++$ ， $++i$ ， $i--$ ， $--i$ ， $*(--i)=*(j++)$ ）

表5.1 PDP-11寻址方式

寻址模式	寻址方式	汇编语法	有效地址EA/操作数S	说明	指令实例
0	寄存器寻址	R_i	$S=(R_i)$	寄存器的值就是操作数	MOV R0, R1 ;R0 <- R1
1	寄存器间接寻址	(R_i)	$EA=(R_i)$	寄存器的值是操作数地址	MOV R0, (R1) ;寄存器 <- 存储器
2	自增寻址	$(R_i)+$	$EA=(R_i), R_i++$	寄存器的值是操作数地址，取数后寄存器自增 (byte +1, word +2)	MOV R0, (SP)+ ;寄存器 <- 存储器
3	自增间接寻址	$@(R_i)+$	$EA=((R_i)), R_i++$	寄存器的值是操作数地址的地址，取数后寄存器自增 (byte +1, word +2)	INC @(R2)+ ;存储器内容加1
4	自减寻址	$-(R_i)$	$EA=(R_i), R_i--$	先将寄存器自减 (byte -1, word -2)，运算结果是操作数地址	MOV -(SP) , R0 ;存储器 <- 寄存器
5	自减间接寻址	$@-(R_i)$	$EA=((R_i)), R_i--$	先将寄存器自减 (byte -1, word -2)，运算结果是操作数地址的地址	INC @-(R2) ;存储器内容加1
6	变址寻址	$index(R_i)$	$EA=(R_i)+index$	操作数地址=寄存器的值+16位index	ADD R0, 200(R1) ;寄存器+存储器，结果送寄存器
7	变址间址寻址	$@index(R_i)$	$EA=((R_i)+index)$	操作数地址的地址=寄存器的值+16位index	ADD R0, @300(R2) ;寄存器+存储器，结果送寄存器

- 当寄存器字段A为111（7）时，也就是为PC寄存器时，可以扩展出4种寻址方式，如表5.2所示
 - 指令CLR @#1100：是将地址为1100的存储单元的内容清零
 - 指令INC 10也可以写成INC 10(PC)：是将地址为PC+10的存储单元的内容加1
 - 指令CLR @10也可以写成CLR @10(PC)：假设地址为PC+10的存储单元的内容为2000H，则该指令是将地址为2000H的存储单元的内容清零

表5.2 PDP-11扩展寻址方式

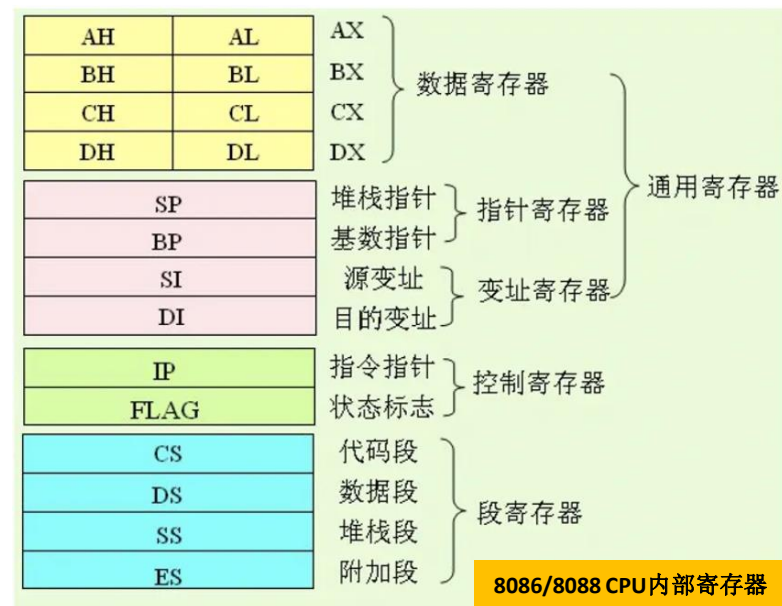
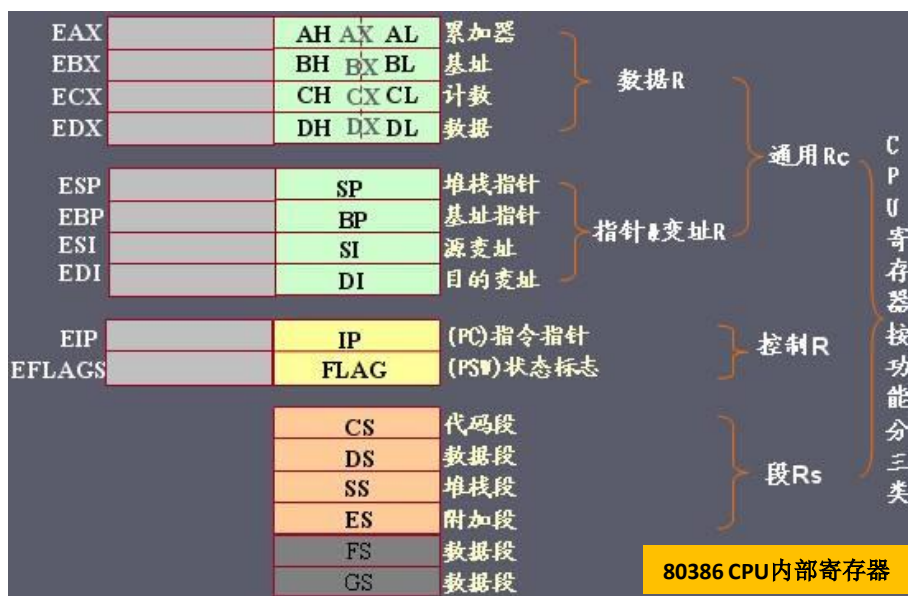
寻址模式	寻址方式	汇编语法	有效地址	功能	指令实例
2	立即数寻址	#n	S=index	操作数是指令字后续一个机器字	ADD #10,R0 ;R0 <- R0+10
3	直接寻址	@#n	EA=index	操作数地址是指令字后续一个机器字	CLR @#1100 ;存储器内容清零
6	相对寻址	A / A(PC)	EA=PC+index	操作数地址=PC+A；A是指令字后续一个机器字	INC 10 / INC 10(PC) ;存储器内容加1
7	相对间接	@A / @A(PC)	EA=(PC+index)	操作数地址的地址=PC+A；PC是下条指令地址	CLR @10 /CLR @10(PC) ;存储器内容清零

• 5.7.2 Intel x86指令系统

- Intel x86指令系统是由16位的8086/8088指令系统发展而来的**32位CISC指令系统**
- **80386 CPU采用32位的Intel x86指令系统，有8个32位通用寄存器、6个16位的段寄存器，以及32位的指令指针（程序计数器）和状态寄存器：**
 - ① 32位通用寄存器：EAX、EBX、ECX、EDX、ESP、EBP、ESI、EDI（表5.3，**书上有误**）
 - ② 32位指令指针（程序计数器）：EIP
 - ③ 32位状态寄存器：EFLAGS
 - ④ 16位段寄存器：CS、DS、SS、ES、FS、GS

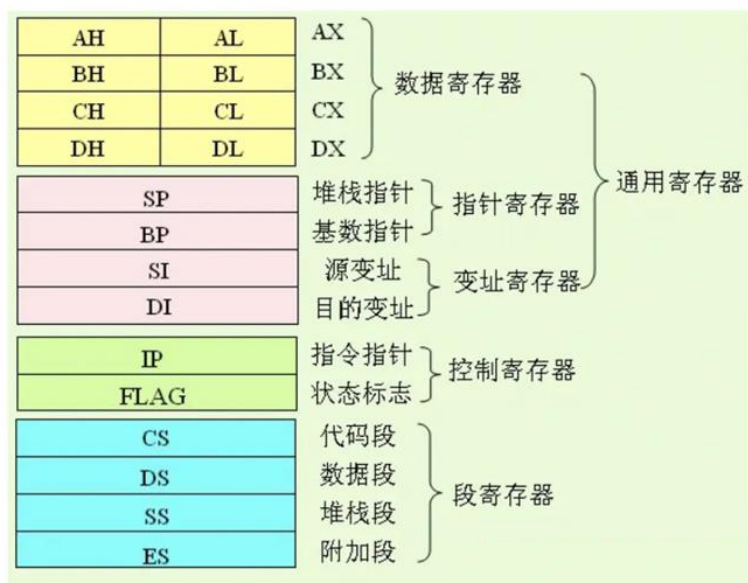
表5.3 寄存器地址编号

寄存器编号	000	001	010	011	100	101	110	111
寄存器名	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

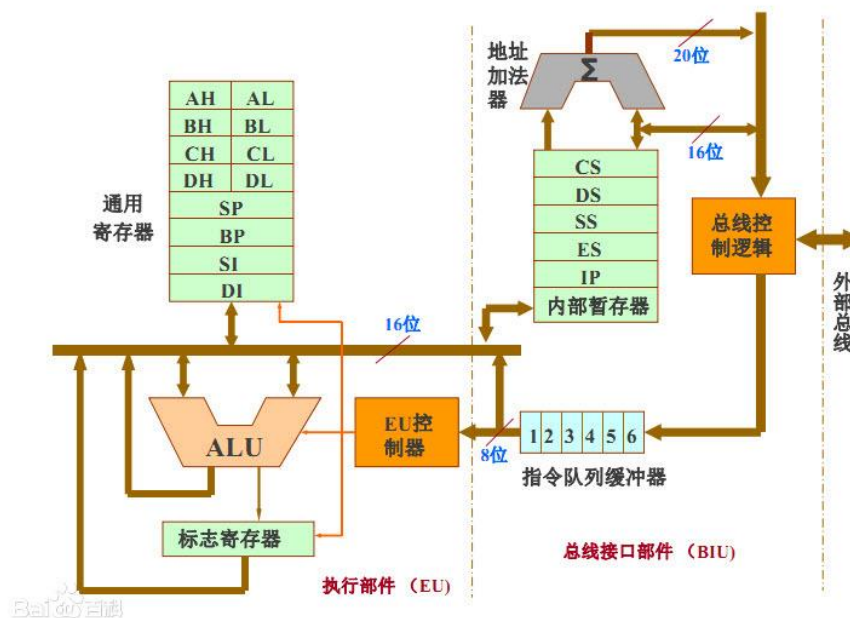


Intel 8086/8088指令系统

- Intel 8086/8088微处理器使用的指令系统，为**16位CISC指令系统**。
- 共有**14个16位寄存器**；最短的指令为**1个字节**，最长的指令为**6个字节**。



8086/8088CPU内部寄存器



8086/8088CPU内部结构

Intel x86-64指令系统

- 首先讲一下什么是**Intel x86**? x86是指Intel的开发的一种**32位**指令集, 从**80386**时代开始, 一直沿用至今, 是一种**CISC**指令集。**x86-64**是x86 CPU开始迈向**64位**的时候, 有两种选择: (1) 向下兼容x86; (2) 完全重新设计指令集, 不兼容x86。
- AMD**抢跑了, 比Intel率先制造出了商用的兼容x86的CPU, AMD称之为**AMD64**, 抢了64位PC的第一桶金, 得到了用户的认同。而Intel选择了设计一种不兼容x86的全新64为指令集, 称之为**IA-64** (安腾), 但是比AMD晚了一步, 而且**IA-64**也挺惨淡的, 因为是全新设计的CPU, 没有编译器, 也不支持Windows。
- 后来不得不在时机落后的情况下也开始支持**AMD64**的指令集, 但是换了个名字, 叫**x86-64**, 表示是x86指令集的64位扩展, 大概是不愿意承认这玩意是AMD设计出来的。也就是说实际上, x86-64、x64、AMD64基本上是同一个东西, 我们现在用的Intel/AMD的桌面级CPU基本上都是**Intel x86-64**。

Intel x86-64 CPU中的寄存器

操作数大小	可用寄存器
8 位	AL、BL、CL、DL、DIL、SIL、BPL、SPL、R8L、R9L、R10L、R11L、R12L、R13L、R14L、R15L
16 位	AX、BX、CX、DX、DI、SI、BP、SP、R8W、R9W、R10W、R11W、R12W、R13W、R14W、R15W
32 位	EAX、EBX、ECX、EDX、EDI、ESI、EBP、ESP、R8D、R9D、R10D、R11D、R12D、R13D、R14D、R15D
64 位	RAX、RBX、RCX、RDX、RDI、RSI、RBP、RSP、R8、R9、R10、R11、R12、R13、R14、R15

– 1、指令格式

- 采用变长指令格式（图5.15），包括：
 - ① 前缀Prefix
 - ② 操作码OP
 - ③ Mod R/M
 - ④ SIB
 - ⑤ 偏移量Disp
 - ⑥ 立即数Imm
- 最短的指令为单字节指令（8位），最长的指令为15字节指令（120位）

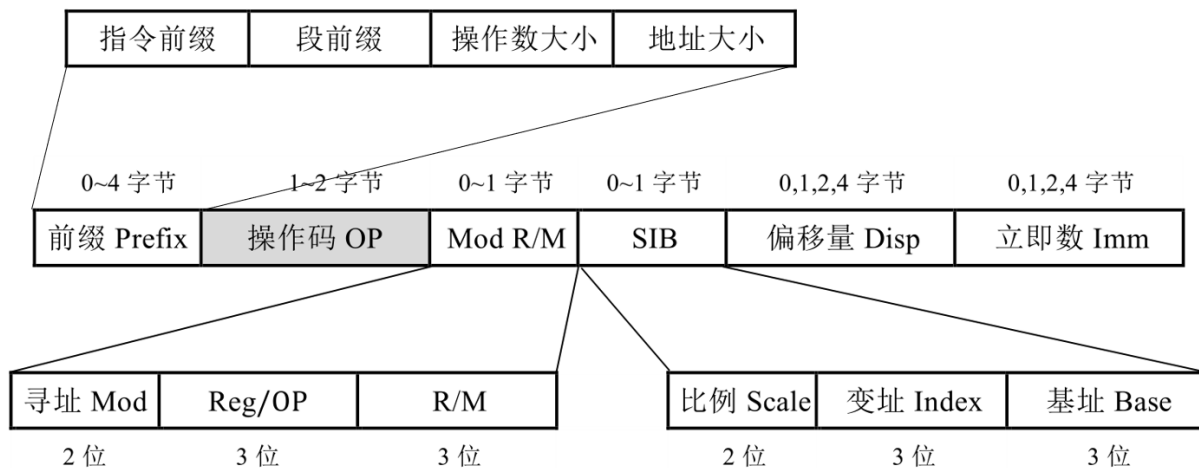


图5.15 Intel x86 指令格式

- (1) 前缀Prefix

- 前缀Prefix字段包括4部分：

- ① 指令前缀（1个字节）：包括锁定前缀和重复前缀两种；锁定前缀（Lock）用于多处理器环境中共享存储器的排他性访问；重复前缀（REPNE/REPZ, REP/REPE/REPZ）用于进行字符串循环处理操作

- ② 段前缀（1个字节）：用来指定使用哪个段寄存器取代默认的段寄存器，共有6个段寄存器（CS、DS、SS、ES、FS、GS）

- ③ 操作数大小（1个字节）：用于在32位和16位操作数之间切换

- ④ 地址大小（1个字节）：用于在32位和16位地址空间之间切换，地址大小决定了指令格式中偏移量的大小和在有效地址计算中生成的偏移量的大小

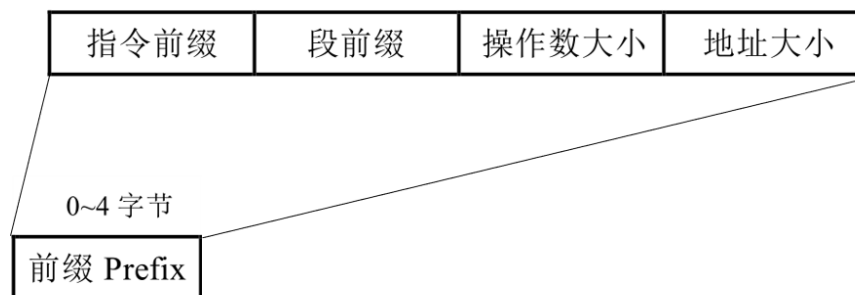


图5.16 前缀Prefix

- (2) 操作码OP

- 图5.17a: 单地址指令; OP=5位; Reg=3位, 表示8个通用寄存器; 如指令push、pop、dec、inc等
- 图5.17b: 双地址指令; OP=6位; d=1, 表示Mod R/M字段中的寄存器操作数为目的寄存器, 否则为源寄存器; w=0, 表示8位操作数, 否则为16位或32位操作数(究竟是16位还是32位由前缀Prefix决定); 如指令add、or、adc、sub、xor、cmp等
- 图5.17c: OP=8位, 可以是零地址、单地址、双地址指令
- 图5.17d: OP=16位, OP的高8位为0FH、低8位为扩展操作码

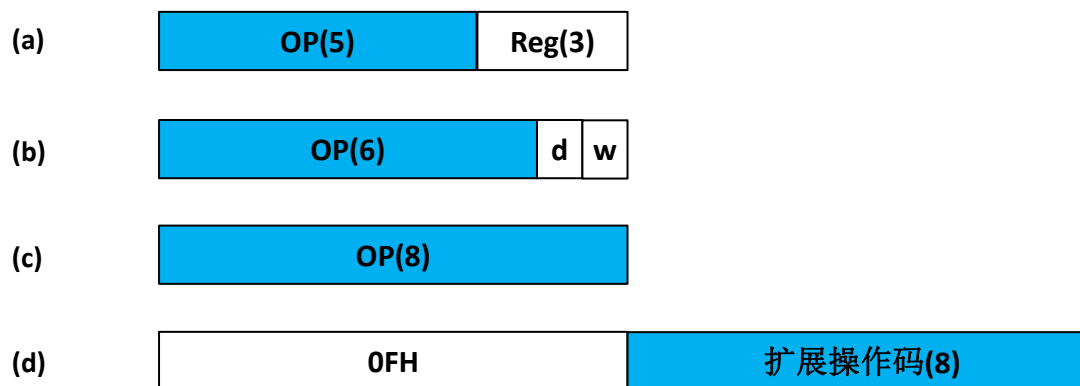


图5.17 Intel x86 指令操作码字段格式

- (3) Mod R/M

- Mod R/M字段用于描述操作数及其寻址方式，Intel x86指令集规定：双操作数最多只能有一个存储器操作数；Mod R/M字段包括3部分：
- 寻址Mod（2位）：与R/M字段一起构成8种寻址方式，如表5.4所示
- Reg/OP（3位）：用于表示寄存器操作数的编号（Reg）；如果是单地址指令，则为操作码的扩展字段（OP）
- R/M（3位）：用于表示另一个操作数，该操作数可能是寄存器（R），也可能是存储器（M）

表5.4 Mod R/M字段对应的寻址方式

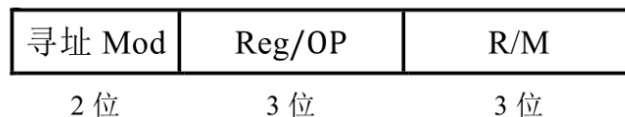


图5.18 Mod R/M字段

Mod	R/M值	操作数类型	寻址方式	有效地址EA/操作数S
00		存储器	寄存器间接寻址	EA=R[R/M]
00	100	存储器	基址+比例变址寻址	EA=SIB
00	101	存储器	偏移量寻址（直接寻址）	EA=Disp32
01		存储器	寄存器相对寻址	EA=R[R/M]+Disp8
01	100	存储器	基址+比例变址+偏移量寻址	EA=SIB+Disp8
10		存储器	寄存器相对寻址	EA=R[R/M]+Disp32
10	100	存储器	基址+比例变址+偏移量变址	EA=SIB+Disp32
11		寄存器	寄存器寻址	S=R[R/M]

- (4) SIB

- SIB (**S**cale, **I**ndex, **B**ase) 字段与Mod R/M字段组合，用于指定寻址方式；SIB字段包括3部分，如图5.19所示：
- **比例Scale** (2位)：指定比例变址中的比例因子，比例因子= 2^{Scale} (Scale=00、01、10、11，对应1、2、4、8四个比例因子)
- **变址Index** (3位)：指定变址寄存器的编号 (8个寄存器)
- **基址Base** (3位)：指定基址寄存器的编号 (8个寄存器)
- SIB字段对应的有效地址： $\text{EA} = \text{R}[\text{Base}] + 2^{\text{Scale}} \times \text{R}[\text{Index}]$

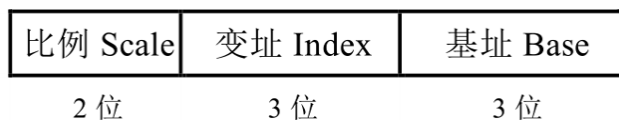


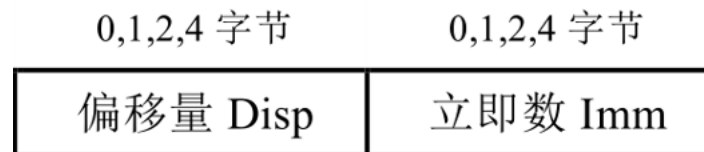
图5.19 SIB字段

- (5) 偏移量Disp

- 偏移量Disp字段可以是8位（1字节）、16位（2字节）、32位（4字节），也可以没有（0字节）
- 偏移量字段是否存在与Mod R/M字段中的寻址方式有关

- (6) 立即数Imm字段

- 立即数Imm字段提供指令所需的8位（1字节）、16位（2字节）、32位（4字节）的立即数操作数，也可以没有（0字节）
- 立即数字段是否存在与指令操作码有关



– 2、指令译码

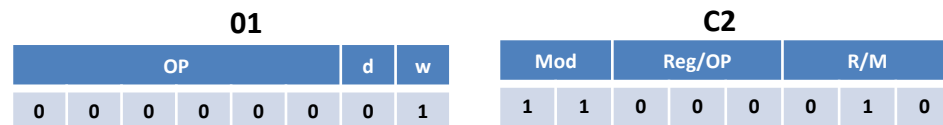
- 指令译码流程包括前缀分析、操作码译码、寻址方式译码等3步：
 - ① **前缀分析**：首先判断有没有前缀Prefix？如果有，则根据不同的前缀值进行指令功能控制（最多有4个字节的前缀）
 - ② **操作码译码**：首先判断是单字节操作码，还是双字节操作码（扩展操作码）？不同的操作码对应不同的指令功能；表5.5为ADD指令的部分操作码
 - ③ **寻址方式译码**：根据Mod R/M字段，确定指令中的操作数和寻址方式，并配合后续的SIB字段、Disp（偏移量）字段、Imm（立即数）字段，计算操作数地址

表5.5 ADD指令部分操作码

操作码	d	w	指令形式	说明
00	0	0	ADD r/m8,r8	d=0，寄存器为源操作数；w=0，操作数为8位
01	0	1	ADD r/m16,r16	w=1，操作数为16位或32位，操作数位宽取决于运行环境和指令前缀
01	0	1	ADD r/m32,r32	
02	1	0	ADD r8,r/m8	d=1，寄存器为目的操作数；w=0，操作数为8位
03	1	1	ADD r16,r/m16	w=1，操作数为16位或32位，操作数位宽取决于运行环境和指令前缀
03	1	1	ADD r32,r/m32	
83	1	1	ADD r/m32,imm8	Reg/OP字段为操作码扩展，值应为000

- 例5.5：假设在一个基于Intel x86指令集的32位运行环境中，有如下指令字：
- （1）**01C2H**；（2）2E 033BH；（3）034C BB66H；（4）8304 BB66H
- 请结合表5.5所示的指令操作码给出对应指令字的汇编代码。

解：



（1）

图5.20 指令字01C2H的指令格式

指令字01C2H的指令格式如图5.20所示：

- 操作码=01，且d=0、w=1，表示为“**ADD r/m32,r32**”指令（32位运行环境）
- Mod=11，表示是寄存器寻址，因此指令为“**ADD r32,r32**”
- Reg/OP=000，表示源操作数为“EAX”；R/M=010，表示目的操作数为“EDX”
- 因此指令字01C2H对应的指令为“**ADD EDX, EAX**”

寄存器编号	000	001	010	011	100	101	110	111
寄存器名	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

表5.5 ADD指令部分操作码

操作码	d	w	指令形式	说明
00	0	0	ADD r/m8,r8	d=0，寄存器为源操作数；w=0，操作数为8位
01	0	1	ADD r/m16,r16	w=1，操作数为16位或32位，操作数位宽取决于运行环境和指令前缀
01	0	1	ADD r/m32,r32	
02	1	0	ADD r8,r/m8	d=1，寄存器为目的操作数；w=0，操作数为8位
03	1	1	ADD r16,r/m16	w=1，操作数为16位或32位，操作数位宽取决于运行环境和指令前缀
03	1	1	ADD r32,r/m32	
83	1	1	ADD r/m32,imm8	Reg/OP字段为操作码扩展，值应为000

表5.4 Mod R/M字段对应的寻址方式

Mod	R/M值	操作数类型	寻址方式	有效地址EA/操作数S
00		存储器	寄存器间接寻址	EA=R[R/M]
00	100	存储器	基址+比例变址寻址	EA=SIB
00	101	存储器	偏移量寻址（直接寻址）	EA=Disp32
01		存储器	寄存器相对寻址	EA=R[R/M]+Disp8
01	100	存储器	基址+比例变址+偏移量寻址	EA=SIB+Disp8
10		存储器	寄存器相对寻址	EA=R[R/M]+Disp32
10	100	存储器	基址+比例变址+偏移量变址	EA=SIB+Disp32
11		寄存器	寄存器寻址	S=R[R/M]

- 例5.5：假设在一个基于Intel x86指令集的32位运行环境中，有如下指令字：
- （1）01C2H；（2）2E 033BH；（3）034C BB66H；（4）8304 BB66H
- 请结合表5.5所示的指令操作码给出对应指令字的汇编代码。

解：

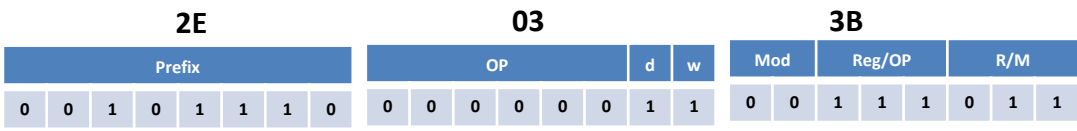


图5.21 指令字2E 033BH的指令格式

- （2）
 - 指令字2E 033BH的指令格式如图5.21所示：

寄存器编号	000	001	010	011	100	101	110	111
寄存器名	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

- Prefix=2E表示段前缀，且是CS段寄存器
- 操作码=03，且d=1、w=1，表示为“ADD r32,r/m32”指令（32位运行环境）
- Mod=00，R/M=011，表示是寄存器间接寻址，因此指令为“ADD r32,m32”；因为R/M=011，表示是EBX寄存器，因此m32为CS:[EBX]（因为有段前缀）
- Reg/OP=111，表示r32为EDI寄存器；因此指令字2E 033BH对应的指令为“ADD EDI, CS:[EBX]”

表5.5 ADD指令部分操作码

操作码	d	w	指令形式	说明
00	0	0	ADD r/m8,r8	d=0，寄存器为源操作数；w=0，操作数为8位
01	0	1	ADD r/m16,r16	w=1，操作数为16位或32位，操作数位宽取决于运行环境和指令前缀
01	0	1	ADD r/m32,r32	
02	1	0	ADD r8,r/m8	d=1，寄存器为目的操作数；w=0，操作数为8位
03	1	1	ADD r16,r/m16	w=1，操作数为16位或32位，操作数位宽取决于运行环境和指令前缀
03	1	1	ADD r32,r/m32	
83	1	1	ADD r/m32,imm8	Reg/OP字段为操作码扩展，值应为000

表5.4 Mod R/M字段对应的寻址方式

Mod	R/M值	操作数类型	寻址方式	有效地址EA/操作数S
00		存储器	寄存器间接寻址	EA=R[R/M]
00	100	存储器	基址+比例变址寻址	EA=SIB
00	101	存储器	偏移量寻址（直接寻址）	EA=Disp32
01		存储器	寄存器相对寻址	EA=R[R/M]+Disp8
01	100	存储器	基址+比例变址+偏移量寻址	EA=SIB+Disp8
10		存储器	寄存器相对寻址	EA=R[R/M]+Disp32
10	100	存储器	基址+比例变址+偏移量变址	EA=SIB+Disp32
11		寄存器	寄存器寻址	S=R[R/M]

- 例5.5：假设在一个基于Intel x86指令集的32位运行环境中，有如下指令字：
- (1) 01C2H； (2) 2E 033BH； (3) 034C BB66H； (4) 8304 BB66H
- 请结合表5.5所示的指令操作码给出对应指令字的汇编代码。

解：

03								4C								BB								66							
OP						d	w	Mod		Reg/OP			R/M			Scale		index			Base			Disp8							
0	0	0	0	0	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	1	0	1	1	0	1	1	0	0	1	1	0

(3)

图5.22 指令字034C BB66H的指令格式

指令字034C BB66H的指令格式如图5.22所示：

- 操作码=03，且d=1、w=1，表示为“ADD r32,r/m32”指令（32位运行环境）
- Mod=01，R/M=100，表示是基址+比例变址+偏移量寻址，因此指令为“ADD r32,m32”
- $EA = SIB + Disp8 = R[Base] + 2^{Scale} \times R[Index] + 66$ ；Base=011，因此R[Base]=EBX；Index=111，因此R[Index]=EDI；Scale=10，因此 $2^{Scale}=4$ ；Disp8=66，因此m32为[EBX+EDI*4+66]
- Reg/OP=001，表示r32为ECX寄存器；因此指令字034C BB66H对应的指令为“ADD ECX, [EBX+EDI*4+66]”

寄存器编号	000	001	010	011	100	101	110	111
寄存器名	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

表5.5 ADD指令部分操作码

表5.4 Mod R/M字段对应的寻址方式

操作码	d	w	指令形式	说明
00	0	0	ADD r/m8,r8	d=0，寄存器为源操作数；w=0，操作数为8位
01	0	1	ADD r/m16,r16	w=1，操作数为16位或32位，操作数位宽取决于运行环境和指令前缀
01	0	1	ADD r/m32,r32	
02	1	0	ADD r8,r/m8	d=1，寄存器为目的操作数；w=0，操作数为8位
03	1	1	ADD r16,r/m16	w=1，操作数为16位或32位，操作数位宽取决于运行环境和指令前缀
03	1	1	ADD r32,r/m32	
83	1	1	ADD r/m32,imm8	Reg/OP字段为操作码扩展，值应为000

Mod	R/M值	操作数类型	寻址方式	有效地址EA/操作数S
00		存储器	寄存器间接寻址	EA=R[R/M]
00	100	存储器	基址+比例变址寻址	EA=SIB
00	101	存储器	偏移量寻址（直接寻址）	EA=Disp32
01		寄存器	寄存器相对寻址	EA=R[R/M]+Disp8
01	100	存储器	基址+比例变址+偏移量寻址	EA=SIB+Disp8
10		存储器	寄存器相对寻址	EA=R[R/M]+Disp32
10	100	存储器	基址+比例变址+偏移量变址	EA=SIB+Disp32
11		寄存器	寄存器寻址	S=R[R/M]

- 解:

- (4)

图5.23 指令字8304 BB66H的指令格式

- | 寄存器编号 | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|
| 寄存器名 | EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |

表5.4 Mod R/M字段对应的寻址方式

Mod	R/M值	操作数类型	寻址方式	有效地址EA/操作数S
00		存储器	寄存器间接寻址	EA=R[R/M]
00	100	存储器	基址+比例变址寻址	EA=SIB
00	101	存储器	偏移量寻址（直接寻址）	EA=Disp32
01		存储器	寄存器相对寻址	EA=R[R/M]+Disp8
01	100	存储器	基址+比例变址+偏移量寻址	EA=SIB+Disp8
10		存储器	寄存器相对寻址	EA=R[R/M]+Disp32
10	100	存储器	基址+比例变址+偏移量变址	EA=SIB+Disp32
11		寄存器	寄存器寻址	S=R[R/M]

– 3、寻址方式

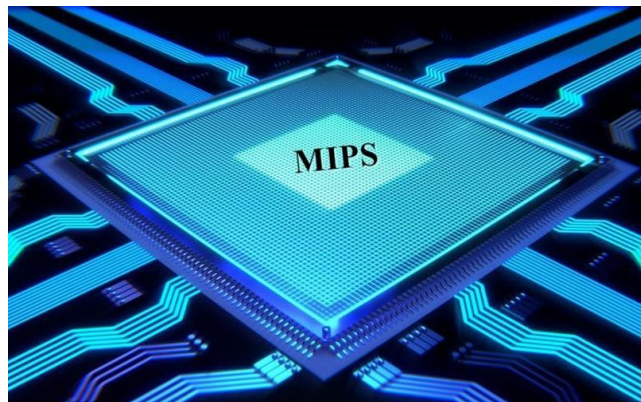
- 操作数的主要来源为：立即数、寄存器、存储单元；当操作数为**存储单元**时，需要进行**地址转换**
- x86处理器采用**段页式**存储管理机制，有3类地址：逻辑地址、线性地址、物理地址
- **逻辑地址**为段式虚拟地址，包括段寄存器和段内偏移地址，即指令中的有效地址EA
- **线性地址**（LA：Linear Address）为页式虚拟地址（见4.6.3 页式虚拟存储器）
- 地址转换时，首先通过分段方式将逻辑地址转换为线性地址，然后再通过分页方式将线性地址转换为**物理地址**
- 在**实模式**下：物理地址（20位）= 段寄存器（16位）左移4位 + 段内偏移地址
- 在**保护模式**下：线性地址（32位）= 段基址（32位）+ 段内偏移地址；再将线性地址通过存储器管理部件（MMU）转换为物理地址（32位）
- 地址转换中最关键的是**段内偏移地址**的计算，段内偏移地址由寻址方式确定，x86的主要寻址方式如表5.6所示

表5.6 x86的主要寻址方式

序号	寻址方式	有效地址EA/操作数S	指令示例
1	立即数寻址	$S = \text{Disp}$	MOV EAX,1000
2	直接寻址	$EA = \text{Disp}$	MOV EAX,[1080H]
3	寄存器寻址	$S = R[R/M]$	MOV EAX,ECX
4	寄存器间接寻址	$EA = R[R/M]$	MOV EAX,[EBX]
5	寄存器相对寻址（基址寻址）	$EA = R[R/M] + \text{Disp8}$	MOV EAX,[ESI+100H]
6	基址+比例变址寻址	$EA = S * \text{index} + \text{Base}$	MOV EAX,[EBX+EDI*4]
7	基址+比例变址+偏移量寻址	$EA = S * \text{index} + \text{Base} + \text{Disp}$	MOV EAX,[EBX+EDI*4+66]
8	相对寻址	$EA = PC + \text{Disp}$	JMP 1000H

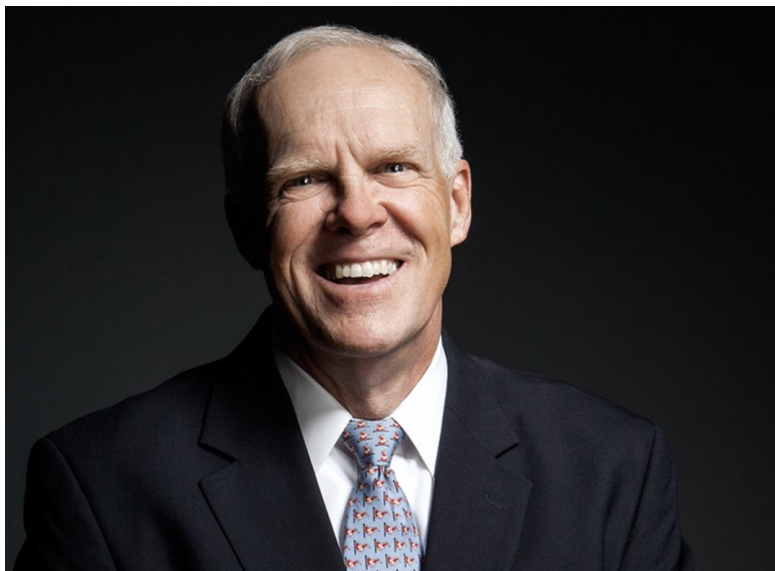
• 5.7.3 MIPS指令系统

- **MIPS: Microprocessor without Interlocked Pipeline Stages**, 没有互锁流水线的微处理器
- **MIPS: Million Instructions Per Second**, 每秒执行百万条指令
- MIPS指令系统是1981年斯坦福大学**Hennessy教授**研究小组研制并商用, 是一种非常优雅、简洁、高效的**RISC体系结构**, 非常适合于教学研究, 包括**MIPS 32** (32位架构) 和**MIPS 64** (64位架构)
- 国产**龙芯**处理器就是基于MIPS指令系统



约翰·亨尼斯（John Hennessy）

- 约翰·亨尼斯（John Hennessy，1953年-），美国计算机科学家。为MIPS科技公司创办人，第十任**斯坦福大学**校长。
- 毕业于维拉诺瓦大学取得电机工程学士学位、纽约石溪大学取得计算机科学硕士及博士学位。
- **1984**年创办MIPS科技公司。**1997**年成为计算机协会会员。**2000**年，得到约翰·冯诺依曼奖。





龙芯



- 2002年8月10日诞生的“龙芯1号”是我国首枚拥有自主知识产权的通用高性能微处理芯片。龙芯从2001年以来共开发了1号、2号、3号三个系列处理器和龙芯桥片系列，在政企、安全、金融、能源等应用场景得到了广泛的应用。**龙芯1号**系列为32位低功耗、低成本处理器，主要面向低端嵌入式和专用应用领域；**龙芯2号**系列为64位低功耗单核或双核系列处理器，主要面向工控和终端等领域；**龙芯3号**系列为64位多核系列处理器，主要面向桌面和服务器等领域。
- 2015年3月31日，中国发射首枚使用“龙芯”北斗卫星。
- 2019年12月24日，龙芯3A4000/3B4000在北京发布，使用与上一代产品相同的28nm工艺，通过设计优化，实现了性能的成倍提升。龙芯坚持自主研发，芯片中的所有功能模块，包括CPU核心等在内的所有源代码均实现自主设计，所有定制模块也均为自主研发。
- 2020年3月3日，360公司与龙芯中科技术有限公司联合宣布，双方将加深多维度合作，在芯片应用和网络安全开发等领域进行研发创新，并展开多方面技术与市场合作。
- 2021年4月龙芯自主指令系统架构（Loongson Architecture，以下简称龙芯架构或LoongArch）的基础架构通过国内第三方知名知识产权评估机构的评估。
- 2021年12月17日，龙芯中科技术股份有限公司（简称“**龙芯中科**”）科创板首发上会。

– 1、MIPS体系结构中的寄存器

- 32个32位的通用寄存器：**\$0 ~ \$31**；也可以表示为：**\$zero**、**\$at**、.....、**\$ra**；见表5.7
- 32个32位单精度浮点寄存器：**f0 ~ f31**；它们可配对为16个64位的浮点寄存器
- 2个特殊的寄存器：**\$hi**、**\$lo**，用于保存乘（64位的乘积）、除（商和余数）运算结果

表5.7 MIPS的通用寄存器

寄存器#	助记符	释义
\$0	\$zero	固定值为0 硬件置位
\$1	\$at	汇编器保留，临时变量
\$2~\$3	\$v0~\$v1	函数调用返回值
\$4~\$7	\$a0~\$a3	4个函数调用参数
\$8~\$15	\$t0~\$t7	暂存寄存器，被调用者按需保存
\$16~\$23	\$s0~\$s7	save寄存器，调用者按需保存
\$24~\$25	\$t8~\$t9	暂存寄存器，同上
\$26~\$27	\$k0~\$k1	操作系统保留，中断异常处理
\$28	<u>\$gp</u>	全局指针 (Global Pointer)
\$29	<u>\$sp</u>	堆栈指针 (Stack Pointer)
\$30	<u>\$fp</u>	帧指针 (Frame Pointer)
\$31	<u>\$ra</u>	函数返回地址 (Return Address)

– 2、MIPS指令格式

- 所有的指令都是**32位的定长指令**（每条指令对应的机器码都是32位），包括**R（寄存器）型**、**I（立即数）型**、**J（无条件转移）型**3种指令格式，见图5.24
- 操作码OP=6位；R型指令的OP=000000，采用扩展操作码的方式，指令的功能由funct字段（6位）决定
- Rs、Rt、Rd为寄存器操作数字段，均为5位，可以访问32个通用寄存器
- shamt字段为移位变量（5位），用于移位指令，其他指令无效
- 立即数有16位（I型指令）和26位（J型指令）两种
- 指令字中没有独立的寻址方式字段，指令的寻址方式由操作码决定

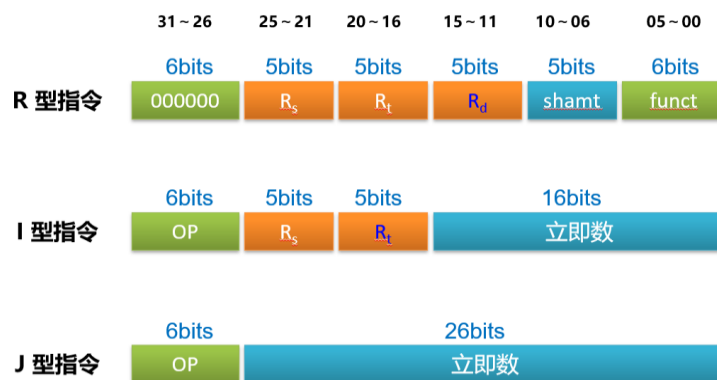


图5.24 MIPS的指令格式

• (1) R型指令（寄存器型）



- R型指令的操作数只能是寄存器，运算结果也只能存入寄存器，属于**RR型（寄存器-寄存器）**指令
- **OP=000000**，指令的功能由**funct**字段指定（6位，64种功能），如表5.8所示
- 如果是双目运算，**rs**、**rt**分别是第一和第二源操作数，**rd**为目的操作数；如果是移位运算，则表示对**rt**的内容进行移位，移位位数由**shamt**指定

表5.8 常用R型指令及其funct编码

funct	指令助记符	指令功能描述	备注
00	sll rd,rt,shamt	R[rd]=R[rt]<<shamt	逻辑左移指令，注意rs字段未使用
02	srl rd,rt,shamt	R[rd]=R[rt]>>shamt	逻辑右移指令，注意rs字段未使用
03	sra rd,rt,shamt	R[rd]=R[rt]>>shamt	算术右移指令，注意rs字段未使用
04	slv rd,rt,rs	R[rd]=R[rt]<<R[rs]	可变左移指令
08	jr rs	PC=R[rs]	无条件跳转指令，R[rs]值应是4的倍数，字对齐
09	jalr rs	R[31]=PC+8 PC=R[rs]	子程序调用指令，R[31]保存程序的断点
12	syscall	系统调用指令	无操作数
16	mfhi rd	R[rd]=HI	取HI寄存器的值指令，mflo指令取LO
17	mthi rs	HI=R[rs]	存HI寄存器的值指令，mtlo指令存LO
24	mult rs,rt	{HI,LO}=R[rs]*R[rt]	有符号乘指令，64位结果送入HI、LO寄存器
32	add rd,rs,rt	R[rd]=R[rs]+R[rt]	加法指令，溢出时发生异常，且不修改R[rd]
34	sub rd,rs,rt	R[rd]=R[rs]-R[rt]	减法指令，溢出时发生异常，且不修改R[rd]
36	and rd,rs,rt	R[rd]=R[rs]&R[rt]	逻辑与指令
37	or rd,rs,rt	R[rd]=R[rs] R[rt]	逻辑或指令
42	slt rd,rs,rt	R[rd]=(R[rs]<R[rt])?1:0	小于置位指令，有符号比较

- 例5.6：根据MIPS指令操作码定义及指令格式，写出下列指令各字段的十进制值：**add \$s1, \$t0, \$s4**。

解：

- add为R型指令：**add rd,rs,rt**；因此OP=000000=0，**func=32**



funct	指令助记符	指令功能描述	备注
00	sll rd,rt,shamt	$R[rd] = R[rt] \ll \text{shamt}$	逻辑左移指令，注意rs字段未使用
02	srl rd,rt,shamt	$R[rd] = R[rt] \gg \text{shamt}$	逻辑右移指令，注意rs字段未使用
03	sra rd,rt,shamt	$R[rd] = R[rt] \gg \text{shamt}$	算术右移指令，注意rs字段未使用
04	slvr rd,rt,rs	$R[rd] = R[rt] \ll R[rs]$	可变左移指令
08	jr rs	$PC = R[rs]$	无条件跳转指令，R[rs]值应是4的倍数，字对齐
09	jalr rs	$R[31] = PC + 4$ $PC = R[rs]$	子程序调用指令，R[31]保存程序的断点
12	syscall	系统调用指令	无操作数
16	mthi rd	$R[rd] = HI$	取HI寄存器的值指令，mthi指令取LO
17	mtlo rs	$HI = R[rs]$	存HI寄存器的值指令，mtlo指令存LO
24	mult rs,rt	$(HI,LO) = R[rs] * R[rt]$	有符号乘指令，64位结果送入HI、LO寄存器
32	add rd,rs,rt	$R[rd] = R[rs] + R[rt]$	加法指令，溢出时发生异常，且不修改R[rd]
34	sub rd,rs,rt	$R[rd] = R[rs] - R[rt]$	减法指令，溢出时发生异常，且不修改R[rd]
36	and rd,rs,rt	$R[rd] = R[rs] \& R[rt]$	逻辑与指令
37	or rd,rs,rt	$R[rd] = R[rs] R[rt]$	逻辑或指令
42	sll rd,rs,rt	$R[rd] = (R[rs] \ll R[rt]) \& 0xFFFFFFFF$	小于字位指令，有符号比较

- 根据MIPS寄存器定义，**rd=\$s1=\$17**，**rs=\$t0=\$8**，**rt=\$s4=\$20**；shamt字段没有使用

- 因此，**add \$s1, \$t0, \$s4**对应的各字段的十进制值如图5.25所示

OP=0	rs=8	rt=20	rd=17	shamt=0	func=32
------	------	-------	-------	---------	---------

表5.7 MIPS的通用寄存器

寄存器#	助记符	释义
\$0	\$zero	固定值为0 硬件置位
\$1	\$at	汇编器保留，临时变量
\$2~\$3	\$v0~\$v1	函数调用返回值
\$4~\$7	\$a0~\$a3	4个函数调用参数
\$8~\$15	\$t0~\$t7	暂存寄存器，被调用者按需保存
\$16~\$23	\$s0~\$s7	save寄存器，调用者按需保存
\$24~\$25	\$t8~\$t9	暂存寄存器，同上
\$26~\$27	\$k0~\$k1	操作系统保留，中断异常处理
\$28	\$gp	全局指针 (Global Pointer)
\$29	\$sp	堆栈指针 (Stack Pointer)
\$30	\$fp	帧指针 (Frame Pointer)
\$31	\$ra	函数返回地址 (Return Address)

图5.25 add指令各字段十进制值

- (2) I型指令（立即数型）



- I型指令就是立即数型指令
- 若是双目运算，则将源操作数 rs 和立即数 imm 分别作为第一和第二操作数，结果送 rt
- 若是Load/Store指令，则将 rs 的值和立即数 imm 的值相加，生成存储单元地址
- 若是条件分支指令，则对 rs 和 rt 的值进行比较，并根据比较结果决定是否转移，转移的目标地址采用相对寻址方式
- 常见的I型指令及其操作码如表5.9所示

表5.9 常用I型指令及其操作码

OP	指令助记符	指令功能描述	备注
04	beq rs,rt,imm	If($R[rs]==R[rt]$) $PC=PC+4+imm<<2$	条件分支指令（相等跳转）
05	bne rs,rt,imm	If($R[rs]!=R[rt]$) $PC=PC+4+imm<<2$	条件分支指令（不等跳转）
08	addi rt,rs,imm	$R[rt]=R[rs]+imm$	立即数加指令，溢出发生异常
10	slti rt,rs,imm	$R[rt]=(R[rs]<imm)?1:0$	小于置位指令，有符号比较
12	andi rt,rs,imm	$R[rt]=R[rs]\&imm$	立即数逻辑与指令
15	lui rt,imm	$R[rt]=imm<<16$	加载立即数指令
35	lw $rt,imm(rs)$	$R[rt]=M[R[rs]+imm]$	取数指令，类似指令还有lb、lh、lbh等
43	sw $rt,imm(rs)$	$M[R[rs]+imm]=R[rt]$	存数指令，类似指令还有sb、sh

- (3) J型指令（无条件转移型）



- J型指令主要是**无条件转移**指令
- 指令中给出的是**26位直接地址（address）**，采用**伪直接寻址方式**；无条件转移的目标地址由PC+4的高4位与address经左移2位后的值拼接得到
- 例如：PC+4=2500 0000H，address=1234567H；PC+4的高4位=2H，address左移2位=01 0010 0011 0100 0101 0110 0111 00= 0100 1000 1101 0001 0101 1001 1100=48D159CH，拼接后的目标地址=248D 159CH
- 常用的J型指令及其操作码如表5.10所示

表5.10 常用J型指令及其操作码

OP	指令助记符	指令功能描述	备注
02	j address	PC <- {(PC+4) _{31:28} ,address,00}	无条件分支指令
03	jal address	R[31] <- PC+8 (无延迟槽+4) PC <- {(PC+4) _{31:28} ,address,00}	子程序调用指令

– 3、MIPS指令寻址方式

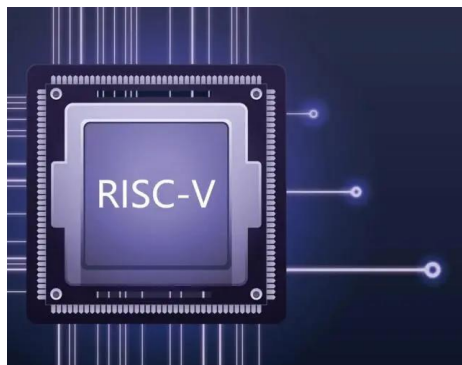
- MIPS指令只有5种寻址方式，如表5.11所示
- R型指令：只有寄存器寻址
- I型指令：有寄存器寻址、立即数寻址、寄存器相对寻址（基址寻址）、相对寻址
- J型指令：只有伪直接寻址

表5.11 MIPS指令的寻址方式

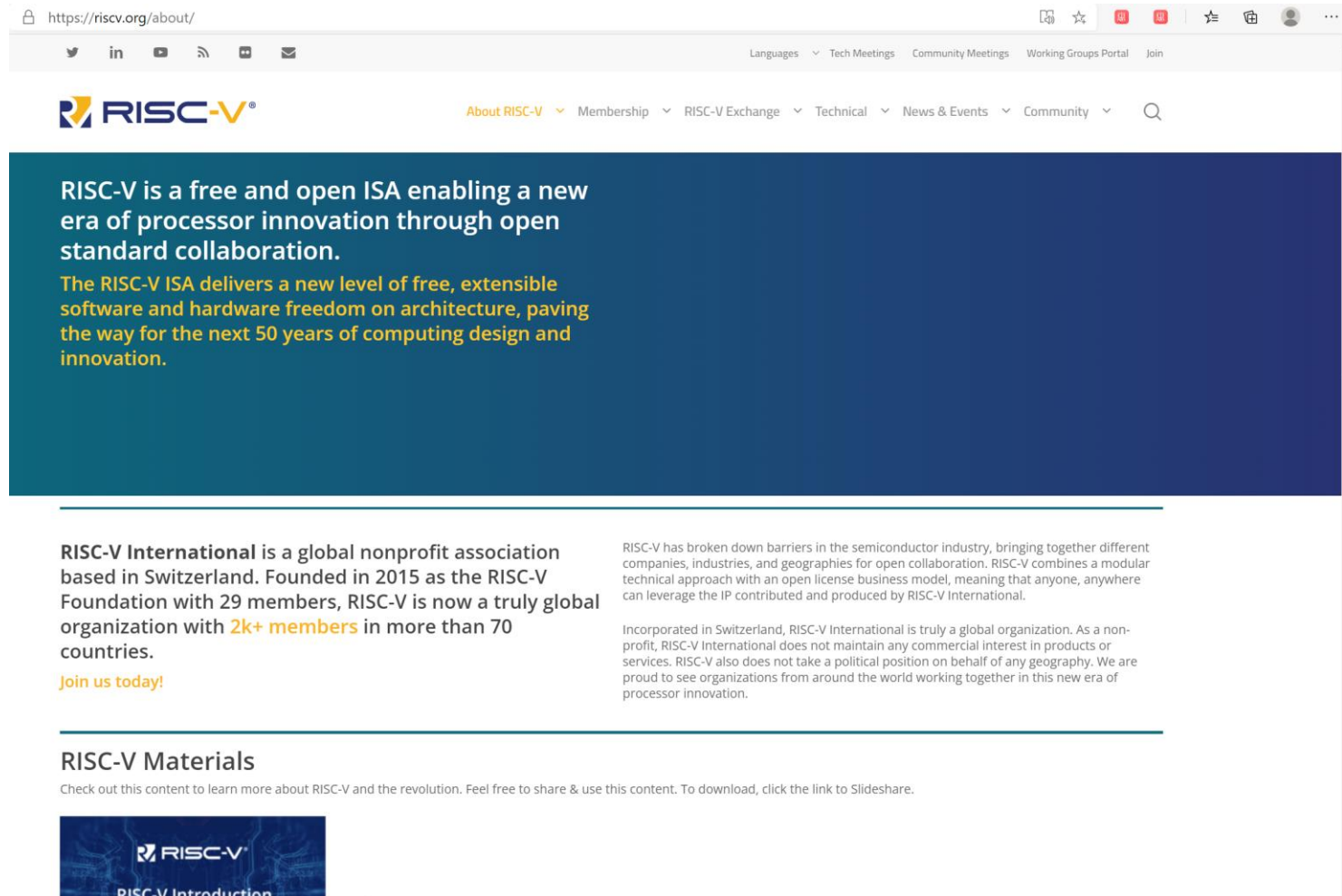
序号	寻址方式	有效地址EA/操作数S	指令示例
1	立即数寻址	$S = \text{imm}$	<code>addi rt,rs,imm</code>
2	寄存器寻址	$S = R[\text{rt}]$	<code>add rd,rs,rt</code>
3	寄存器相对寻址（基址寻址）	$EA = R[\text{rs}] + \text{imm}$	<code>lw rt,imm(rs)</code>
4	相对寻址	$EA = PC + 4 + \text{Disp}$	<code>beq rs,rt,imm</code>
5	伪直接寻址	$EA = \{(PC + 4)_{31:28}, \text{address}, 00\}$	<code>j address</code>

• 5.7.4 RISC-V指令系统

- RISC-V（发音：RISC Five）是第五代RISC指令系统，其结合了ARM、MIPS等RISC指令系统的优势，是完全从零开始重新设计的一款开源指令系统，由RISC-V基金会负责运营
- RISC-V指令集采用模块化的方式设计，包括32位指令集和64位指令集
- 32位RISC-V指令集（RV32G）包括核心指令集RV32I，以及4个标准扩展集：RV32M（乘除法）、RV32F（单精度浮点数）、RV32D（双精度浮点数）、RV32A（原子操作）
- 核心指令集RV32I：只有47条指令，其指令格式规整，易于硬件的实现
- RISC-V模块化的指令集设计方法，方便进行灵活定制与扩展，既可以用于嵌入式微处理器（MCU：Microcontroller Unit，微控制单元），也适合构造服务器、家用电器、工业控制以及传感器中的微处理器（CPU）
- 阿里巴巴旗下的平头哥半导体有限公司的玄铁处理器就采用了RISC-V架构



RISC-V基金会（<https://riscv.org/>）



The image is a screenshot of the RISC-V website's 'About' page. The browser's address bar shows 'https://riscv.org/about/'. The website's header includes the RISC-V logo, a navigation menu with links like 'About RISC-V', 'Membership', 'RISC-V Exchange', 'Technical', 'News & Events', and 'Community', and a search icon. The main content area has a dark blue background with white and yellow text. It states that RISC-V is a free and open ISA enabling processor innovation through open standard collaboration. It also mentions that the RISC-V ISA delivers a new level of free, extensible software and hardware freedom on architecture, paving the way for the next 50 years of computing design and innovation. Below this, there is a section titled 'RISC-V International' which describes it as a global nonprofit association based in Switzerland, founded in 2015 as the RISC-V Foundation with 29 members, and now a truly global organization with 2k+ members in more than 70 countries. A 'Join us today!' link is provided. To the right, there is a paragraph explaining that RISC-V has broken down barriers in the semiconductor industry, bringing together different companies, industries, and geographies for open collaboration. Below this, another paragraph states that RISC-V International is truly a global organization, does not maintain any commercial interest in products or services, and does not take a political position on behalf of any geography. At the bottom, there is a section titled 'RISC-V Materials' with a link to check out content to learn more about RISC-V and the revolution, and a link to download content. A small video player thumbnail for 'RISC-V Introduction' is visible at the bottom left.

<https://riscv.org/about/>

RISC-V

About RISC-V Membership RISC-V Exchange Technical News & Events Community

RISC-V is a free and open ISA enabling a new era of processor innovation through open standard collaboration.

The RISC-V ISA delivers a new level of free, extensible software and hardware freedom on architecture, paving the way for the next 50 years of computing design and innovation.

RISC-V International is a global nonprofit association based in Switzerland. Founded in 2015 as the RISC-V Foundation with 29 members, RISC-V is now a truly global organization with **2k+ members** in more than 70 countries.

[Join us today!](#)

RISC-V has broken down barriers in the semiconductor industry, bringing together different companies, industries, and geographies for open collaboration. RISC-V combines a modular technical approach with an open license business model, meaning that anyone, anywhere can leverage the IP contributed and produced by RISC-V International.

Incorporated in Switzerland, RISC-V International is truly a global organization. As a non-profit, RISC-V International does not maintain any commercial interest in products or services. RISC-V also does not take a political position on behalf of any geography. We are proud to see organizations from around the world working together in this new era of processor innovation.

RISC-V Materials

Check out this content to learn more about RISC-V and the revolution. Feel free to share & use this content. To download, click the link to Slideshare.

RISC-V

RISC-V Introduction

平头哥和玄铁

- **平头哥**半导体有限公司于**2018年10月31日**成立，法定代表人刘湘雯，是阿里巴巴旗下半导体公司。**2019年9月25日**，阿里巴巴在**2019云栖大会**上正式对外发布含光**800AI**芯片。平头哥半导体将打造面向汽车、家电、工业等诸多行业领域的物联网芯片平台。前期由阿里扶持，后期平头哥将会独立化运作，自负盈亏。
- **玄铁910**是平头哥发布的基于**RISC-V**的处理器IP核，开发者可以免费下载**FPGA**代码，开展芯片原型设计架构创新。**2019年7月25日**，平头哥首个产品玄铁**910**正式发布。



– 1、RISC-V通用寄存器

- 32个32位的通用寄存器：**x0 ~ x31**；也可以表示为：**zero、ra、.....、t6**；见表5.12

表5.12 RISC-V的通用寄存器

编号	助记符	英文全称	功能描述
x0	zero	zero	恒零值，可用0号寄存器参与的加法指令实现MOV指令
x1	ra	Return Address	返回地址
x2	sp	Stack Pointer	栈指针，指向栈顶
x3	gp	Global Pointer	全局指针
x4	tp	Thread Pointer	线程寄存器
x5 ~ x7	t0 ~ t2	Temporaries	临时变量，调用者保存寄存器
x8	s0/fp	Saved Register/Frame Pointer	通用寄存器，被调用者保存寄存器，在子程序中使用时必须先压栈保存原值，使用后应出栈恢复原值
x9	s1	Saved Registers	通用寄存器，被调用者保存寄存器
x10 ~ x11	a0 ~ a1	Arguments/Return values	用于存储子程序参数或返回值
x12 ~ x17	a2 ~ a7	Arguments	用于存储子程序参数
x18 ~ x27	s2 ~ s11	Saved Registers	通用寄存器，被调用者保存寄存器
x28 ~ x31	t3 ~ t6	Temporaries	临时变量

– 2、RISC-V指令格式

- RISC-V也是**定长指令**（32位），但其操作码字段预留了扩展空间，可以扩展为变长指令，但指令字长必须是双字节对齐
- RISC-V有**6种指令格式**，如图5.26所示
- 操作码opcode=7位，funct3（3位）、funct7（7位）为扩展操作码；rs1、rs2是源操作数寄存器，rd是目的操作数寄存器，都是5位，用于选择32个寄存器，且位置固定；其余位置用于填充立即数字段，因此有些格式的指令（S型、B型、J型）立即数字段分散在几个地方
- RISC-V也没有寻址方式字段，寻址方式由操作码决定；RISC-V指令字中各字段位置固定，可以提高指令译码的速度，硬件实现也更容易

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0								
funct7				rs2				rs1			funct3			rd			opcode		R-type	10条		
imm[11:0]								rs1			funct3			rd			opcode		I-type	25条		
imm[11:5]				rs2				rs1			funct3			imm[4:0]			opcode		S-type	3条		
imm[12]		imm[10:5]			rs2				rs1			funct3			imm[4:1]		imm[11]		opcode		B-type	6条
imm[31:12]										rd					opcode		U-type	2条				
imm[20]		imm[10:1]				imm[11]			imm[19:12]				rd			opcode		J-type	1条			

图5.26 RISC-V的指令格式

合计47条

- (1) R型指令（寄存器型）

- R型指令的操作数只能是寄存器（rs1、rs2），运算结果也只能存入寄存器（rd），属于RR（寄存器-寄存器）型指令
- 主操作码OP=33H，指令的功能由funct3（3位）、funct7（7位）字段指定，如图5.27所示
- 核心指令集RV32I中有10条R型指令，如表5.13所示

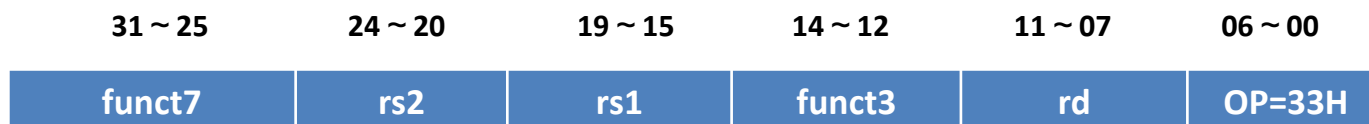


图5.27 R型指令

表5.13 RISC-V的R型指令

类型	指令示例	功能描述	同类指令
算术逻辑运算指令	add rd,rs1,rs2	$R[rd] = R[rs1] + R[rs2]$	add、sub、xor、or、and
关系运算指令	slt rd,rs1,rs2	$R[rd] = (R[rs1] < R[rs2]) ? 1 : 0$	slt（有符号）、sltu（无符号）
移位指令	sll rd,rs1,rs2	$R[rd] = R[rs1] \ll R[rs2]$	sll、srl、sra

- (2) I型指令（立即数型）

- I型（立即数型）指令的操作数为两个寄存器（rs1、rd）和立即数（12位）
- 操作码包括主操作码OP（7位）以及扩展操作码funct3（3位），如图5.28所示
- 核心指令集RV32I中有25条R型指令，如表5.14所示



图5.28 I型指令

表5.14 RISC-V的I型指令

类型	指令示例	功能描述	同类指令
算术逻辑运算指令	addi rd,rs1,imm	$R[rd]=R[rs1] + imm$	addi、xori、ori、andi
关系运算指令	slti rd,rs1,imm	$R[rd]=R[rs1] < imm$	slti、sltiu
移位指令	slli rd,rs1,imm	$R[rd]=R[rs1] \ll imm$	slli、srli、srai
访存指令（取数）	lw rd,imm(rs1)	$R[rd]=M[R[rs1] + imm]$	lw、lb、lbu、lh、lhu
系统控制类指令	jalu rd,rs1,imm	$PC=R[rs1] + imm$ $R[rd]=PC+4$	jalu（跳转指令）
系统控制类指令	ecall	系统调用	ecall、fence、fence.i、ebreak
特权指令	csrrw rd,csr,rs1	$R[rd]=csr$ $csr=R[rs1]$	csrrw、csrrs、csrrc、csrrwi、csrrsi、csrrci

- (3) S型指令（写存储器型）
 - 写存储器型指令由于不存在目的寄存器rd，因此不能使用I型指令的格式，只能单独设置一个S型指令格式；其操作数包括两个寄存器rs1、rs2，以及立即数imm（两部分，共12位）
 - 操作码包括主操作码OP（7位）以及扩展操作码funct3（3位），如图5.29所示
 - 核心指令集RV32I中有3条S型指令，如表5.15所示

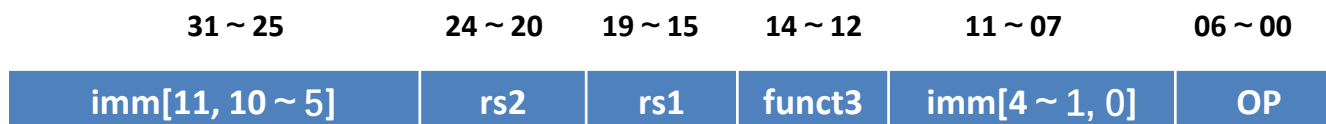


图5.29 S型指令

表5.15 RISC-V的S型指令

类型	指令示例	功能描述	同类指令
访存指令（存数）	sw rs2,imm(rs1)	$M[R[rs1] + imm] = R[rs2]$	sw、sb、sh

- (4) B型指令（条件分支型）

- B型指令用于表示条件分支指令，B型指令也不存在目的寄存器rd，其指令格式与S型指令类似；其操作数包括两个寄存器rs1、rs2，以及立即数imm（两部分，共12位，与S型指令略有不同）；B型指令也称为SB型指令
- 操作码包括主操作码OP（7位）以及扩展操作码funct3（3位），如图5.30所示
- 核心指令集RV32I中有6条B型指令，如表5.16所示

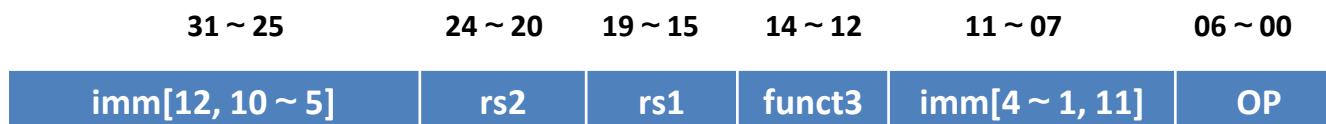


图5.30 B型指令

表5.16 RISC-V的B型指令

类型	指令示例	功能描述	同类指令
分支指令（条件转移）	beq rs1,rs2,imm	if([R[rs1] == R[rs2]) PC=PC+imm<<1	beq、bne、blt、bge、bltu、bgeu

- (5) U型指令（更大立即数型）
 - I型指令的立即数只有12位，范围较小；U型指令的立即数有20位，范围较大；U型指令只有一个目的寄存器rd
 - 操作码只有主操作码OP（7位），如图5.31所示
 - 核心指令集RV32I中有2条U型指令，如表5.17所示



图5.31 U型指令

表5.17 RISC-V的U型指令

类型	指令示例	功能描述
立即数加载	lui rd,imm	$R[rd] = \text{imm} \ll 12$
立即数加载	auipc rd,imm	$R[rd] = PC + \text{imm} \ll 12$

- (6) J型指令（无条件跳转型）

- J型指令用于实现无条件跳转，其操作数包括一个目的寄存器rd和20位立即数；J型指令与U类指令格式类似，也称为UJ型指令
- 操作码只有主操作码OP（7位），如图5.32所示
- 核心指令集RV32I中有1条J型指令，如表5.18所示

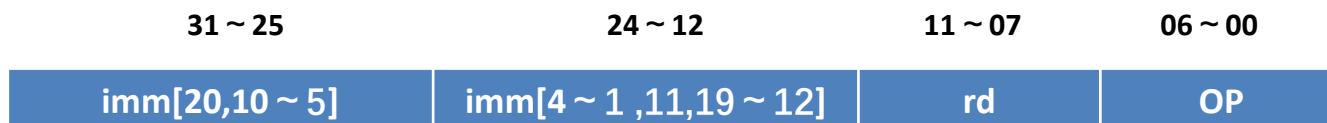


图5.32 J型指令

表5.18 RISC-V的J型指令

类型	指令示例	功能描述
子程序调用	jal rd,imm	PC=PC+imm<<1 R[rd] <- PC+4 rd为x1时可实现子程序调用；rd=x0时，可实现无条件跳转

Base Integer Instructions: RV32I and RV64I										RV Privileged Instructions													
Category	Name	Fmt	RV32I Base					+RV64I					Category	Name	Fmt	RV mnemonic							
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2				SLLW	rd,rs1,rs2				Trap Mach-mode trap return	R		MRET							
	Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt				SLLIW	rd,rs1,shamt				Supervisor-mode trap return	R		SRET							
	Shift Right Logical	R	SRL	rd,rs1,rs2				SRLW	rd,rs1,rs2				Interrupt Wait for Interrupt	R		WFI							
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt				SRLIW	rd,rs1,shamt				MMU Virtual Memory FENCE	R		SFENCE.VMA	rs1,rs2						
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2				SRAW	rd,rs1,rs2				Examples of the 60 RV Pseudoinstructions										
Arithmetic	Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt				SRAIW	rd,rs1,shamt				Branch = 0 (BEQ rs,x0,imm)	J		BEQZ	rs,imm						
	ADD	R	ADD	rd,rs1,rs2				ADDW	rd,rs1,rs2				Jump (uses JAL x0,imm)	J		J	imm						
	ADD Immediate	I	ADDI	rd,rs1,imm				ADDIW	rd,rs1,imm				MoVe (uses ADDI rd,rs,0)	R		MV	rd,rs						
	SUBtract	R	SUB	rd,rs1,rs2				SUBW	rd,rs1,rs2				RETurn (uses JALR x0,0,rs)	I		RET							
	Load Upper Imm	U	LUI	rd,imm									Optional Compressed (16-bit) Instruction Extension: RV32C										
Add Upper Imm to PC																							
Logical	XOR	R	XOR	rd,rs1,rs2				Loads	Load Word	CL	C.LW	rd',rs1',imm	LW	rd',rs1',imm*4									
	XOR Immediate	I	XORI	rd,rs1,imm					Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4									
	OR	R	OR	rd,rs1,rs2					Float Load Word SP	CL	C.FLW	rd',rs1',imm	FLW	rd',rs1',imm*8									
	OR Immediate	I	ORI	rd,rs1,imm					Float Load Word	CI	C.FLWSP	rd,imm	FLW	rd,sp,imm*8									
	AND	R	AND	rd,rs1,rs2					Float Load Double	CL	C.FLD	rd',rs1',imm	FLD	rd',rs1',imm*16									
Compare	AND Immediate	I	ANDI	rd,rs1,imm					Float Load Double SP	CI	C.FLDSP	rd,imm	FLD	rd,sp,imm*16									
	Set <	R	SLT	rd,rs1,rs2				Stores	Store Word	CS	C.SW	rs1',rs2',imm	SW	rs1',rs2',imm*4									
	Set < Immediate	I	SLTI	rd,rs1,imm					Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4									
	Set < Unsigned	R	SLTU	rd,rs1,rs2					Float Store Word SP	CSS	C.FSWSP	rs2,imm	FSW	rs2,sp,imm*8									
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm					Float Store Double	CS	C.FSD	rs1',rs2',imm	FSD	rs1',rs2',imm*16									
Branches	Branch =	B	BEQ	rs1,rs2,imm					Float Store Double SP	CSS	C.FSDSP	rs2,imm	FSD	rs2,sp,imm*16									
	Branch ≠	B	BNE	rs1,rs2,imm					Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1								
	Branch <	B	BLT	rs1,rs2,imm					ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm									
	Branch ≥	B	BGE	rs1,rs2,imm					ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16									
	Branch < Unsigned	B	BLTU	rs1,rs2,imm					ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4									
Jump & Link	Branch ≥ Imm Unsigned	B	BGEU	rs1,rs2,imm					SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1									
	J&L	J	JAL	rd,imm					AND	CR	C.AND	rd,rs1	AND	rd,rd,rs1									
Jump & Link Register	Jump & Link Register	I	JALR	rd,rs1,imm					AND Immediate	CI	C.ANDI	rd,imm	ANDI	rd,rd,imm									
									OR	CR	C.OR	rd,rs1	OR	rd,rd,rs1									
Synch	Synch thread	I	FENCE						eXclusive OR	CR	C.XOR	rd,rs1	AND	rd,rd,rs1									
	Synch Instr & Data	I	FENCE.I						MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0									
Environment	CALL	I	ECALL						Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm									
	BREAK	I	EBREAK						Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm									
Control Status Register (CSR)										Shifts Shift Left Imm													
Control Status Register (CSR)	Read/Write	I	CSRRW	rd,csr,rs1									CI	C.SLLI	rd,imm	SLLI	rd,rd,imm						
	Read & Set Bit	I	CSRRS	rd,csr,rs1									CI	C.SRAI	rd,imm	SRAI	rd,rd,imm						
	Read & Clear Bit	I	CSRRC	rd,csr,rs1									CI	C.SRLI	rd,imm	SRLI	rd,rd,imm						
	Read/Write Imm	I	CSRRWI	rd,csr,imm																			
	Read & Set Bit Imm	I	CSRRSI	rd,csr,imm																			
Loads	Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm																			
	Load Byte	I	LB	rd,rs1,imm																			
	Load Halfword	I	LH	rd,rs1,imm																			
	Load Byte Unsigned	I	LBU	rd,rs1,imm																			
	Load Half Unsigned	I	LHU	rd,rs1,imm																			
Stores	Load Word	I	LW	rd,rs1,imm																			
	Store Byte	S	SB	rs1,rs2,imm																			
	Store Halfword	S	SH	rs1,rs2,imm																			
	Store Word	S	SW	rs1,rs2,imm																			
+RV64I										Optional Compressed Extension: RV64C													
LWU										All RV32C (except C.JAL, 4 word loads, 4 word stores) plus:													
LD										ADD Word (C.ADDW)													
										Load Doubleword (C.LD)													
										ADD Imm. Word (C.ADDIW)													
										Load Doubleword SP (C.LDSP)													
										SUBtract Word (C.SUBW)													
										Store Doubleword (C.SD)													
										Store Doubleword SP (C.SDSP)													
32-bit Instruction Formats																							
R	31	27	26	25	24	20	19	15	14	12	11	7	6	0									
I	funct7					rs2					funct3					rd							
S	imm[11:0]					rs1					funct3					rd							
B	imm[11:5]					rs2					rs1					imm[4:0]							
U	imm[12:10:5]					rs2					rs1					funct3							
J	imm[31:12]					rs1					funct3					imm[4:11]							
	imm[20:10:11:19:12]					rd					rd					opcode							
16-bit (RVC) Instruction Formats																							
CR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
CI	funct4					rd/rs1					rs2					op							
CSS	funct3					imm					rd/rs1					imm							
CIW	funct3					imm					imm					rd'							
CL	funct3					imm					rs1'					rd'							
CS	funct3					imm					rs1'					rs2'							
CB	funct3					offset					rs1'					offset							
CJ	funct3					jump target																	

RISC-V Integer Base (RV32I/64I), privileged, and optional RV32C/64C. Registers x1-x31 and the PC are 32 bits wide in RV32I and 64 in RV64I (x0=0). RV64I adds 12 instructions for the wider data. Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction.

- **RISC-V核心指令集RV32I的47条指令：**
 - 移位指令（6条）：**sll、slli、srl、srli、sra、srai**
 - 算术运算指令（5条）：**add、addi、sub、lui、adipc**
 - 逻辑运算指令（6条）：**xor、xori、or、ori、and、andi**
 - 比较指令（4条）：**slt、slti、sltu、sltiu**
 - 分支指令（条件转移指令）（6条）：**beq、bne、blt、bge、bltu、bgeu**
 - 跳转指令（无条件转移指令）（2条）：**jal、jalr**
 - 同步指令（2条）：**fence、fence.i**
 - 环境指令（2条）：**ecall、ebreak**
 - 控制状态寄存器读写指令（6条）：**csrrw、csrrs、csrrc、csrrwi、csrrsi、csrrci**
 - 取数指令（5条）：**lb、lh、lbu、lhu、lw**
 - 存数指令（3条）：**sb、sh、sw**

– 3、RISC-V寻址方式

- RISC-V比MIPS少了伪直接寻址方式，只有4种寻址方式，如表5.19所示

表5.19 RISC-V指令的寻址方式

序号	寻址方式	有效地址EA/操作数S	指令示例
1	立即数寻址	$S = \text{imm}$	<code>addi rd,rs1,imm</code>
2	寄存器寻址	$S = R[\text{rs1}]$	<code>add rd,rs1,rs2</code>
3	寄存器相对寻址（基址寻址）	$EA = R[\text{rs1}] + \text{imm}$	<code>lw rd,imm(rs1)</code>
4	相对寻址	$EA = PC + \text{imm} \ll 1$	<code>beq rs1,rs2,imm</code>

本章小结

- 机器指令
- 指令系统（也称为指令集）
- 一个完善的指令系统应该满足下面的要求：完备性、规整性、有效性、兼容性、可扩展性

- 指令格式：指令包括操作码OP和地址码A



图5.2 指令的一般格式

- 定长指令系统（如RISC计算机）

CISC: Complex Instruction Set Computer, 复杂指令系统计算机

- 变长指令系统（如CISC计算机）

RISC: Reduced Instruction Set Computer, 精简指令系统计算机

- 定长操作码、变长操作码、扩展操作码技术

- 三地址指令、双地址指令、单地址指令、零地址指令

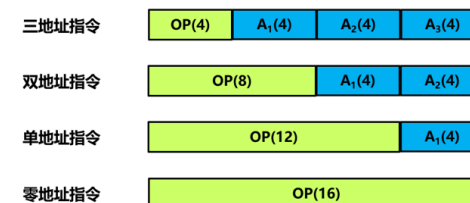


图5.3 扩展操作码

- 指令寻址方式：顺序寻址方式、跳跃寻址方式（用于转移指令等）
- （重点）操作数寻址方式：**立即寻址、直接寻址、寄存器寻址、间接寻址、寄存器间接寻址、相对寻址、变址寻址、基址寻址、堆栈寻址、其他寻址（变址+间接寻址，间接+变址寻址，相对+间接寻址）等



图5.5 含寻址方式字段的单地址指令格式

- 指令类型：算术逻辑运算指令、移位操作指令、数据传送指令、堆栈操作指令、字符串处理指令、程序控制指令（转移指令、循环控制指令、子程序调用与返回指令）、输入输出指令、其他指令



- 指令格式设计:

- ① 指令编码格式的设计: 定长指令格式 (RISC)、变长指令格式 (CISC)、混合指令格式
- ② 操作码的设计
- ③ 地址码的设计
- ④ 寻址方式的设计

- CISC: Complex Instruction Set Computer, 复杂指令系统计算机

- (重点) CISC的特点:

- ① 指令系统复杂庞大, 指令数目多达200-300条
- ② 寻址方式多
- ③ 指令格式多
- ④ 指令字长不固定
- ⑤ 对访存指令不加限制
- ⑥ 各种指令使用频率相差大
- ⑦ 各种指令执行时间相差大
- ⑧ 大多数采用微程序控制器

- RISC: Reduced Instruction Set Computer, 精简指令系统计算机

- (重点) RISC的特点:

- ① 优先选择使用频率最高的一些简单指令, 以及一些很有用但不复杂的指令, 避免使用复杂指令
- ② 大多数指令在一个时钟周期内执行完成
- ③ 采用LOAD/STORE结构, 只允许LOAD/STORE指令访问主存, 其余指令只能对寄存器操作数进行处理
- ④ 采用简单的指令格式和寻址方式, 指令长度固定
- ⑤ 固定的指令格式; 指令长度、格式固定, 可简化指令的译码逻辑, 有利于提高流水线的执行效率; 为了便于编译的优化, 常采用三地址指令格式
- ⑥ 面向寄存器的结构; 为减少访问主存, CPU内应设大量的通用寄存器
- ⑦ 采用硬布线控制逻辑; 由于指令系统的精简, 控制部件可由组合逻辑实现, 不用或少用微程序控制, 这样可使控制部件的速度大大提高
- ⑧ 注重编译的优化, 力求有效地支持高级语言程序

• PDP-11指令系统

- PDP-11是DEC公司在1970年推出的一款经典的16位小型计算机，是IBM PC出现之前最为流行的小型计算机
- PDP-11指令系统属于典型的**16位CISC指令系统**，指令字长通常为单字长（16位），有单操作数（单地址）、双操作数（双地址）两种指令格式
- 双操作数时，通过采用变址寻址方式，可以增加1~2个立即数index字段，从而将指令扩展为双字长（32位）、三字长（48位）指令格式
- PDP-11的寄存器字段A为3位，可寻址**8个寄存器**；寻址特征字段I为3位，有**8种寻址方式**

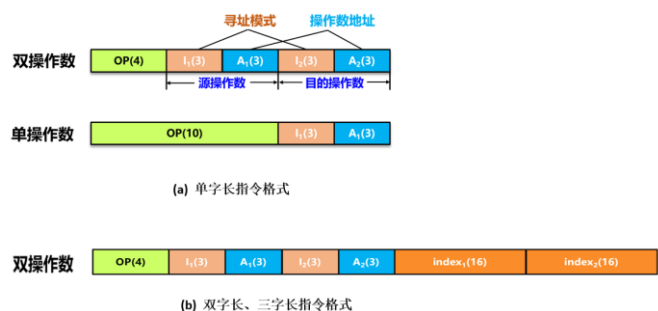


图5.14 PDP-11指令格式

表5.1 PDP-11寻址方式

寻址模式	寻址方式	汇编语法	有效地址EA/操作数S	说明	指令实例
0	寄存器寻址	R_i	$S=(R_i)$	寄存器的值就是操作数	MOV R0,R1 ;R0 ← R1
1	寄存器间接寻址	(R_i)	$EA=(R_i)$	寄存器的值是操作数地址	MOV R0,(R1) ;寄存器 ← 存储器
2	自增寻址	$(R_i)+$	$EA=(R_i), R_i++$	寄存器的值是操作数地址，取数后寄存器自增(byte+1,word+2)	MOV R0,(SP)+ ;寄存器 ← 存储器
3	自增间接寻址	$@(R_i)+$	$EA=((R_i)), R_i++$	寄存器的值是操作数地址的地址，取数后寄存器自增(byte+1,word+2)	INC @(R2)+ ;存储器内容加1
4	自减寻址	$-(R_i)$	$EA=(R_i), R_i--$	先将寄存器自减(byte-1,word-2)，运算结果是操作数地址	MOV -(SP),R0 ;存储器 ← 寄存器
5	自减间接寻址	$@-(R_i)$	$EA=((R_i)), R_i--$	先将寄存器自减(byte-1,word-2)，运算结果是操作数地址的地址	INC @-(R2) ;存储器内容加1
6	变址寻址	$index(R_i)$	$EA=(R_i)+index$	操作数地址=寄存器的值+16位index	ADD R0,200(R1) ;寄存器+存储器，结果送寄存器
7	变址间址寻址	$@index(R_i)$	$EA=((R_i)+index)$	操作数地址的地址=寄存器的值+16位index	ADD R0,@300(R2) ;寄存器+存储器，结果送寄存器

• Intel x86指令系统

- Intel x86指令系统是由16位的8086/8088指令系统发展而来的**32位CISC指令系统**
- 80386 CPU**采用32位的Intel x86指令系统，有8个32位通用寄存器、6个16位的段寄存器，以及32位的状态寄存器和程序计数器（指令指针）
- 采用**变长指令格式**，包括：前缀Prefix、操作码OP、Mod R/M、SIB、偏移量Disp、立即数Imm等字段；最短的指令为单字节指令（8位），最长的指令为**15字节指令**（120位）
- 指令译码流程包括指令前缀分析、操作码译码、寻址方式译码等3步
- x86处理器采用**段页式**存储管理机制：
 - 在**实模式**下：物理地址（20位）= 段寄存器（16位）左移4位 + 段内偏移地址
 - 在**保护模式**下：先计算线性地址（32位）= 段基址（32位）+ 段内偏移地址；再将线性地址通过存储器管理部件（MMU）转换为物理地址（32位）
 - 地址转换中最关键的是**段内偏移地址**的计算，段内偏移地址由**寻址方式**确定，x86的主要有8种寻址方式

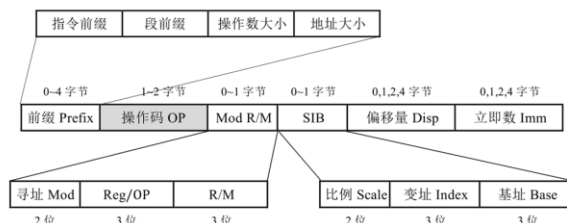
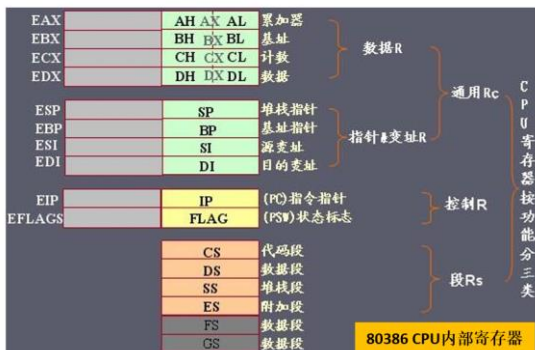


图5.15 Intel x86 指令格式

表5.6 x86的主要寻址方式

序号	寻址方式	有效地址EA/操作数S	指令示例
1	立即数寻址	S=Disp	MOV EAX,1000
2	直接寻址	EA=Disp	MOV EAX,[1080H]
3	寄存器寻址	S=R[R/M]	MOV EAX,ECX
4	寄存器间接寻址	EA=R[R/M]	MOV EAX,[EBX]
5	寄存器相对寻址（基址寻址）	EA=R[R/M]+Disp8	MOV EAX,[ESI+100H]
6	基址+比例变量寻址	EA=S*index+Base	MOV EAX,[EBX+EDI*4]
7	基址+比例变量+偏移量寻址	EA=S*index+Base+Disp	MOV EAX,[EBX+EDI*4+66]
8	相对寻址	EA=PC+Disp	JMP 1000H

• (重点) MIPS指令系统

- MIPS: Microprocessor without Interlocked Pipeline Stages, 没有互锁流水线的微处理器; MIPS: Million Instructions Per Second, 每秒执行百万条指令
- MIPS指令系统是1981年斯坦福大学Hennessy教授研究小组研制并商用, 是一种非常优雅、简洁、高效的RISC体系结构, 非常适合于教学研究, 包括MIPS 32 (32位架构) 和MIPS 64 (64位架构)
- MIPS处理器有32个32位的通用寄存器: $\$0 \sim \31 ; 32个32位单精度浮点寄存器: $f0 \sim f31$, 它们可配对为16个64位的浮点寄存器; 2个特殊的寄存器: $\$hi$ 、 $\$lo$, 用于保存乘 (64位的乘积)、除 (商和余数) 运算结果
- 所有的指令都是32位的定长指令, 包括R型 (寄存器型)、I型 (立即数型)、J型 (无条件转移型) 3种指令格式
- MIPS指令只有5种寻址方式; R型指令: 只有寄存器寻址; I型指令: 有寄存器寻址、立即数寻址、基址寻址 (寄存器相对寻址)、相对寻址; J型指令: 只有伪直接寻址

表5.7 MIPS的通用寄存器

寄存器#	助记符	释义
\$0	\$zero	固定值为0 硬件置位
\$1	\$at	汇编器保留, 临时变量
\$2~\$3	\$v0~\$v1	函数调用返回值
\$4~\$7	\$a0~\$a3	4个函数调用参数
\$8~\$15	\$t0~\$t7	暂存寄存器, 被调用者按需保存
\$16~\$23	\$s0~\$s7	save寄存器, 调用者按需保存
\$24~\$25	\$t8~\$t9	暂存寄存器, 同上
\$26~\$27	\$k0~\$k1	操作系统保留, 中断异常处理
\$28	\$gp	全局指针 (Global Pointer)
\$29	\$sp	堆栈指针 (Stack Pointer)
\$30	\$fp	帧指针 (Frame Pointer)
\$31	\$ra	函数返回地址 (Return Address)

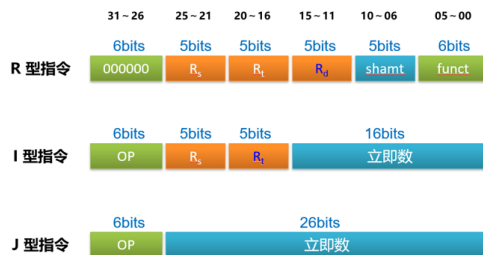


图5.24 MIPS的指令格式

表5.11 MIPS指令的寻址方式

序号	寻址方式	有效地址EA/操作数S	指令示例
1	立即数寻址	S=imm	addi rt,rs,imm
2	寄存器寻址	S=R[rt]	add rd,rs,rt
3	寄存器相对寻址 (基址寻址)	EA=R[rs]+imm	lw rt,imm(rs)
4	相对寻址	EA=PC+4+Disp	beq rs,rt,imm
5	伪直接寻址	EA=((PC+4) _{31:28} ,address,00)	j address

• （重点）RISC-V指令系统

- RISC-V（发音：RISC Five）是**第五代RISC指令系统**，其结合了ARM、MIPS等RISC指令系统的优势，是完全从零开始重新设计的一款开源指令系统，由RISC-V基金会负责运营
- RISC-V指令集采用**模块化**的方式设计，**32位RISC-V指令集（RV32G）**包括核心指令集**RV32I**，以及4个标准扩展集：**RV32M**（乘法）、**RV32F**（单精度浮点数）、**RV32D**（双精度浮点数）、**RV32A**（原子操作）
- 核心指令集**RV32I**：只有**47条指令**，其指令格式规整，易于硬件的实现
- RISC-V处理器有**32个32位的通用寄存器：x0 ~ x31**
- RISC-V也是**定长指令（32位）**，RISC-V有**6种指令格式**
- RISC-V比MIPS少了伪直接寻址方式，只有**4种寻址方式**

表5.12 RISC-V的通用寄存器

编号	助记符	英文名称	功能描述
x0	zero	zero	恒零值，可用0号寄存器参与的加法指令实现MOV指令
x1	ra	Return Address	返回地址
x2	sp	Stack Pointer	栈指针，指向栈顶
x3	gp	Global Pointer	全局指针
x4	tp	Thread Pointer	线程寄存器
x5 ~ x7	t0 ~ t2	Temporaries	临时变量，调用者保存寄存器
x8	s0/fp	Saved Register/Frame Pointer	通用寄存器，被调用者保存寄存器，在子程序中使用时必须先压栈保存原值，使用后应出栈恢复原值
x9	s1	Saved Registers	通用寄存器，被调用者保存寄存器
x10 ~ x11	a0 ~ a1	Arguments/Return values	用于存储子程序参数或返回值
x12 ~ x17	a2 ~ a7	Arguments	用于存储子程序参数
x18 ~ x27	s2 ~ s11	Saved Registers	通用寄存器，被调用者保存寄存器
x28 ~ x31	t3 ~ t6	Temporaries	临时变量

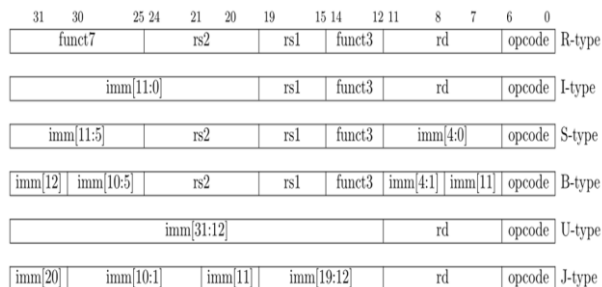


图5.26 RISC-V的指令格式

表5.19 RISC-V指令的寻址方式

序号	寻址方式	有效地址EA/操作数S	指令示例
1	立即数寻址	S=imm	addi rd,rs1,imm
2	寄存器寻址	S=R[rs1]	add rd,rs1,rs2
3	寄存器相对寻址（基址寻址）	EA=R[rs1]+imm	lw rd,imm(rs1)
4	相对寻址	EA=PC+imm<<1	beq rs1,rs2,imm

习题 (P182-185)

- 5.2
- 5.3
- 5.4
- 5.5
- 5.7
- 5.8
- 5.10
- 5.11
- 5.12

关于作业提交

- **1周内**必须提交（上传到学院的FTP服务器上），否则认为是迟交作业；如果期末仍然没有提交，则认为是未提交作业
 - 作业完成情况成绩=第1次作业提交情况*第1次作业评分+第2次作业提交情况*第2次作业评分+.....+第N次作业提交情况*第N次作业评分
 - 作业评分：A（好）、B（中）、C（差）三挡
 - 作业提交情况：按时提交（1.0）、迟交（0.5）、未提交（0.0）
- 请采用电子版的格式（**PPT文档**）上传到FTP服务器上，文件名取“学号+姓名+第X次作业.ppt”
 - 例如：30620192203840+孙明策+第5次作业.ppt
- 第5次作业提交的截止日期为：**2022年4月28日晚上24点。**

实践训练（实验4）

- 在**MIPS**汇编器**MARS**中，利用最少的**MIPS**指令编写一个内存数据冒泡排序程序
- 在**RISC-V**汇编器**RARS**中，利用最少的**RISC-V**指令编写一个内存数据冒泡排序程序

Thanks