

The background of the entire image is a dark blue, textured surface with a complex, glowing circuit board pattern. In the center-right, a square microchip is depicted with a translucent, glowing blue casing. Inside the casing, intricate circuitry is visible, and a central square area is highlighted with a grid of red and green dots. The chip is mounted on a base with numerous small, metallic pins or solder points extending downwards. The overall lighting is a deep blue, creating a high-tech, digital atmosphere.

计算机组成原理



计算机组成原理

五、指令系统



|| 本章主要内容

- **指令系统概述**
- 指令格式
- 寻址方式
- RISC 与 CISC
- MIPS指令系统
- 其他指令系统

指令系统基本概念

■ 机器指令（指令）

- 计算机能直接识别、执行的某种操作命令

■ 指令系统（指令集）

- 一台计算机中所有机器指令的集合

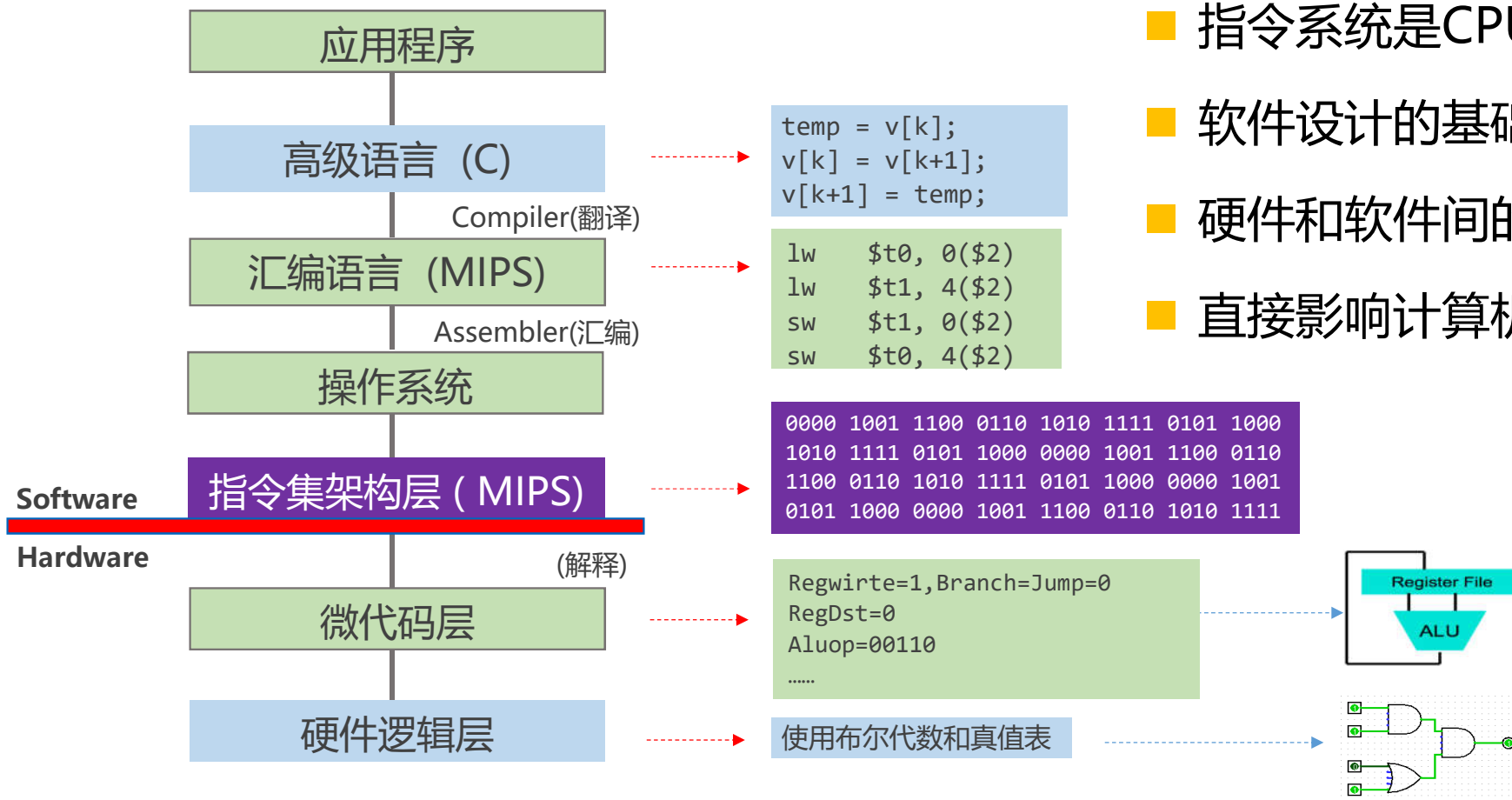
■ 系列机

- 基本指令系统相同，基本系统结构相同的计算机

- ◆ IBM, PDP-11, VAX-11, Intel-x86

- 解决软件兼容的问题

计算机指令系统层次



- 指令系统是CPU设计的依据
- 软件设计的基础
- 硬件和软件间的界面
- 直接影响计算机系统性能

指令系统设计要求

- 完备性：指令丰富，功能齐全，使用方便
- 有效性：程序占空间小，执行速度快
- 规整性：
 - 对称性 （对不同寻址方式的支持）
 - 匀齐性 （对不同数据类型的支持）
 - 一致性 （指令长度和数据长度的一致性）
- 兼容性：系列机软件向上兼容

|| 本章主要内容

- 指令系统概述
- **指令格式**
- 寻址方式
- RISC 与 CISC
- MIPS指令系统
- 其他指令系统

指令格式

■ 表示一条指令的机器字，称为**指令字**，简称**指令**

■ **指令格式**：用二进制代码表示指令的结构形式

□ 指令要求计算机处理什么数据？

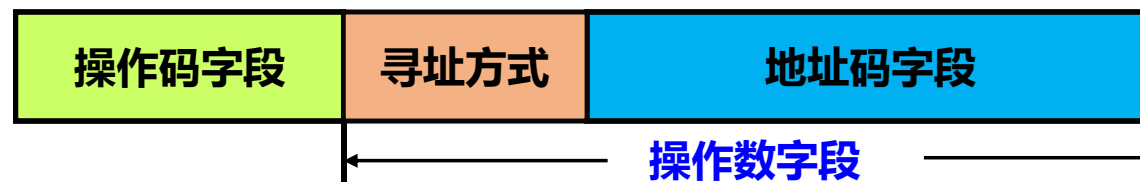
指令的操作数需要解决的问题

□ 指令要求计算机对数据做什么处理？

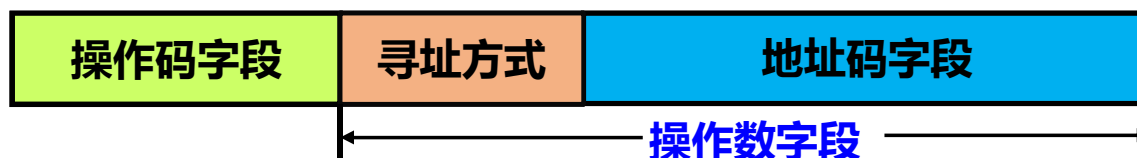
□ 计算机怎样才能得到要处理的数据？

指令的操作码需要解决的问题

指令的寻址方式需要解决的问题



操作码(OP)与地址码(AC)



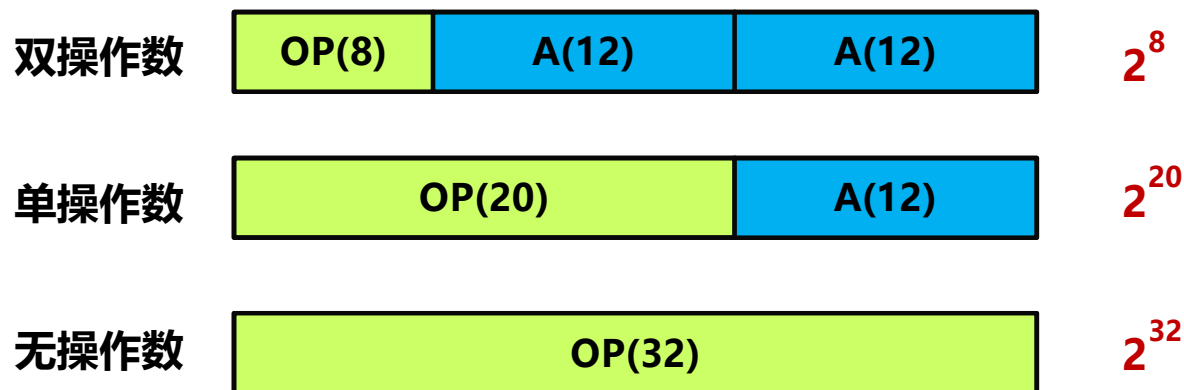
■ 操作码字段长度决定指令系统规模

- 每条指令对应一个操作码
- **定长操作码** $\text{Length}_{\text{OP}} = \lceil \log_2 n \rceil$
- **变长操作码** 操作码向不用的地址码字段扩展

■ 操作数字段可能有多

- **寻址方式字段** 长度与寻址方式种类有关，也可能隐含在操作码字段
- **地址码字段** 作用及影响、长度和寻址方式有关

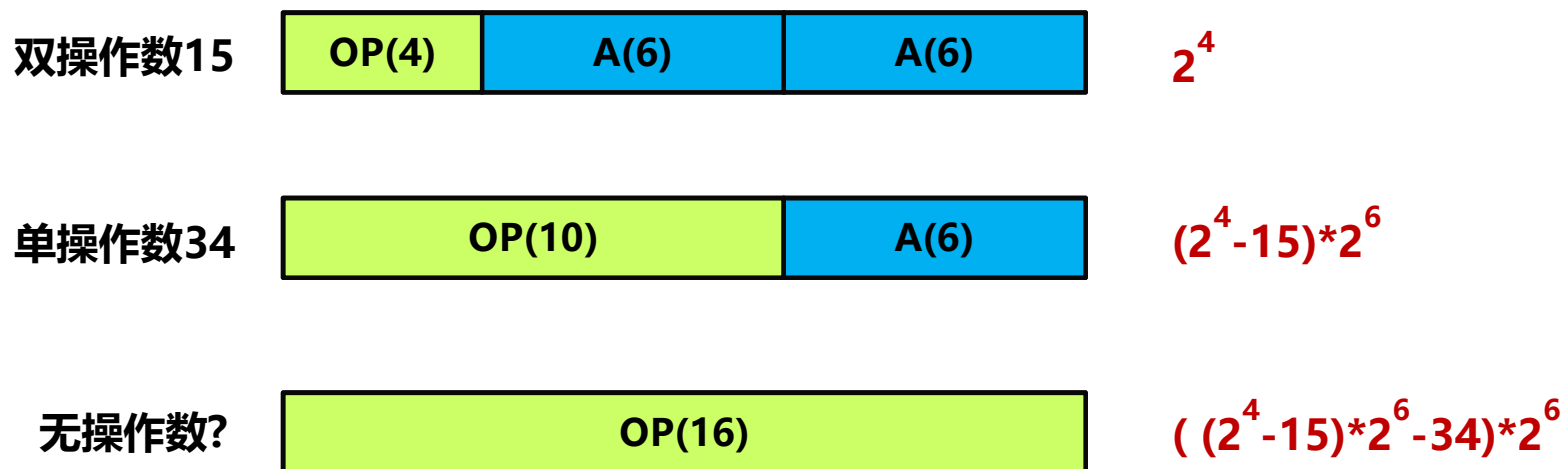
扩展操作码



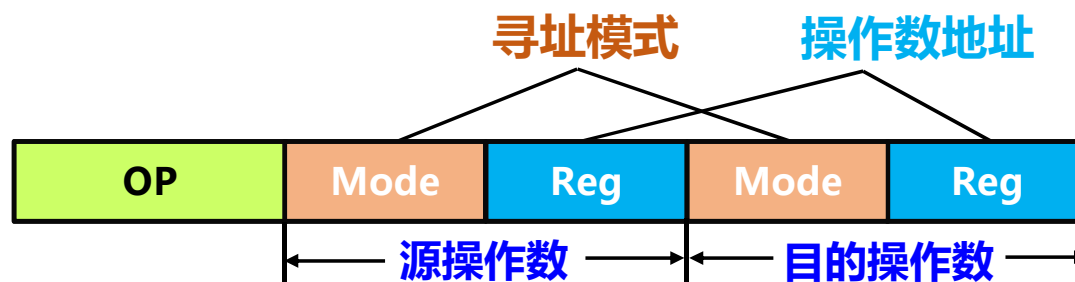
- 3种指令操作码部分不得重叠，否则无法区分，译码
- 设双操作数指令数为 k ，显然 $k < 2^8$
- $2^8 - k$ 为多余状态，可用于表示其他类型指令
- 可用于单操作数指令的条数 = $(2^8 - k) * 2^{12}$ ， 2^{12} 是多余12位组合

扩展指令举例

- 设某指令系统指令字长16位，每个地址码为6位。若要求设计二地址指令15条、一地址指令34条，问最多还可设计多少条零地址指令？



指令格式设计



- 根据指令规模及是否支持操作码扩展，确定操作码字段长度
- 根据对操作数的要求确定地址码字段的个数
- 根据寻址方式的要求，为各地址码字段确定寻址方式字段长度
- 定长还是变长

指令格式设计举例

■ **例1.** 字长16位，主存64K，指令单字长单地址，80条指令。寻址方式有直接、间接、相对、变址。请设计指令格式。

- 80条指令 \Rightarrow OP字段需7位 ($2^7=128$)
- 4种寻址方式 \Rightarrow 寻址方式位需2位
- 单字长单地址 \Rightarrow 地址码长度 $= 16 - 7 - 2 = 7$ 位



指令分类方法

■ 按计算机系统的层次结构分类

- 微指令、机器指令、宏指令

■ 按操作数物理位置分类

- 存储器 - 存储器 (SS) 型、寄存器 - 寄存器 (RR) 型、寄存器 - 存储器 (RS) 型

■ 按指令长度分类

- 定长指令, 变长指令

■ 按操作数个数分类

- 四地址、三地址、二地址、单地址、零地址

■ 按指令功能分类

按指令字长度分类

- **指令字长度**：指令中包含二进制代码的位数
- 字长与机器字的长度有关：**单字长，双字长，半字长**
 - 指令字越长，地址码长度越长，可直接寻址空间越大
 - 指令字越长，占用空间越大，取指令越慢
- **等长指令**：结构简单，控制线路简单，MIPS指令
- **变长指令**：结构灵活，充分利用指令长度，控制复杂，X86指令



按操作数个数分类

三地址指令



$(A1) \text{ OP } (A2) \rightarrow A3$

二地址指令



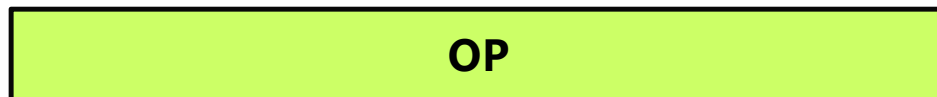
$(A1) \text{ OP } (A2) \rightarrow A1$

单地址指令



$(AC) \text{ OP } (A1) \rightarrow AC$

零地址指令



如停机、空操作、开关中断等

按功能分类 8088/8086典型指令

■ 数据传送类

- 取数 MOV AX, TEMP
- 存数 MOV TEMP, AX
- 传送 MOV AX, CX

■ 算术运算类

- 定点 +, -, ×, ÷ 等
- 浮点 +, -, ×, ÷ 求反, 求补等

■ 逻辑运算类

- NOT, AND, OR, XOR, TEST

■ 程序控制类

- 无条件转移 JMP 条件转移 C, Z, N, P, V
- 转子程序 JSR 子程序返回 RET 中断返回 IRET

■ 输入/输出类

- IN AX, n OUT n, AX

■ 其他类

- 标志操作: CLC (clear carry flag)
- CLI (clear interrupt enable flag)
- HLT, WAIT

|| 本章主要内容

- 指令系统概述
- 指令格式
- **寻址方式**
- RISC 与 CISC
- MIPS指令系统
- 其他指令系统

寻址方式

■ 寻找指令或操作数有效地址的方式

□ 指令寻址

- ◆ 顺序寻址

- ◆ 跳跃寻址

□ 操作数寻址

- ◆ 立即寻址、直接寻址

- ◆ 间接寻址、寄存器寻址

- ◆ 寄存器间接寻址、相对寻址

- ◆ 基址\变址寻址、复合寻址

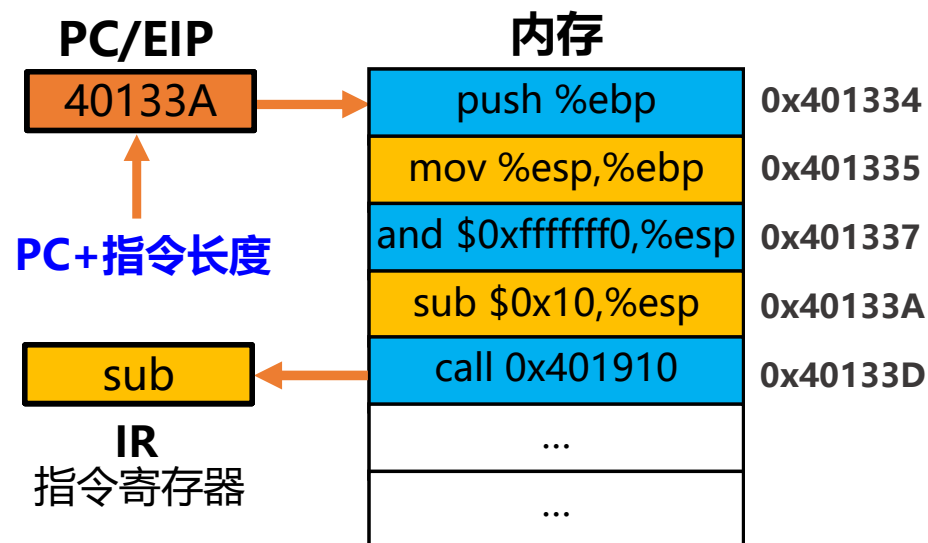
顺序寻址

顺序寻址方式

- 程序对应的机器指令序列在主存顺序存放
- 执行时从第一条指令开始，逐条取出并执行

实现方式

- 程序计数器（PC）对指令序号进行计数
- PC存放下条指令地址，初始值为程序首址
- 执行一条指令， $PC = PC + \text{当前指令字节长度}$

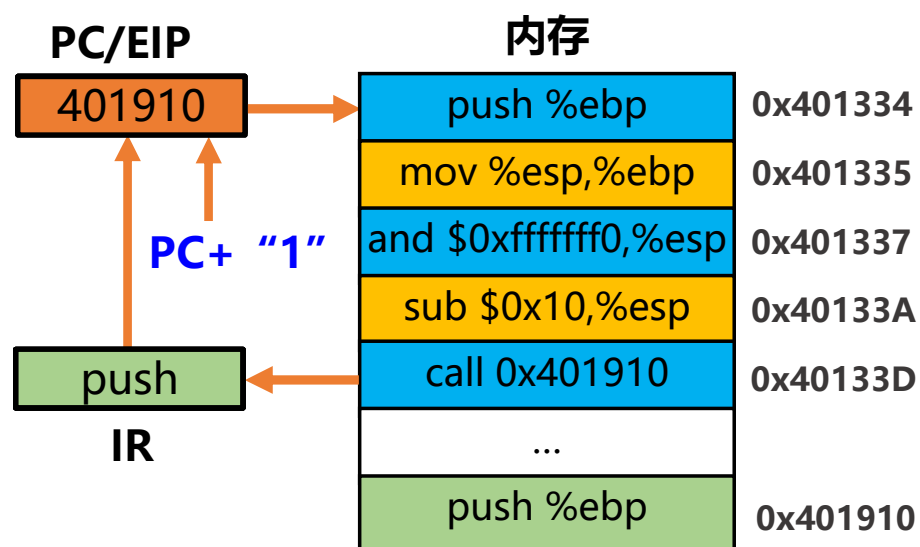


Mem[pc++] \rightarrow IR

跳跃寻址

■ **跳跃寻址方式**：当程序中出现分支或循环时，就会改变程序的执行顺序

- 下条指令地址不是PC++得到，而是由指令本身给出
- 跳跃的处理方式是重新修改PC的内容，然后进入取指令阶段

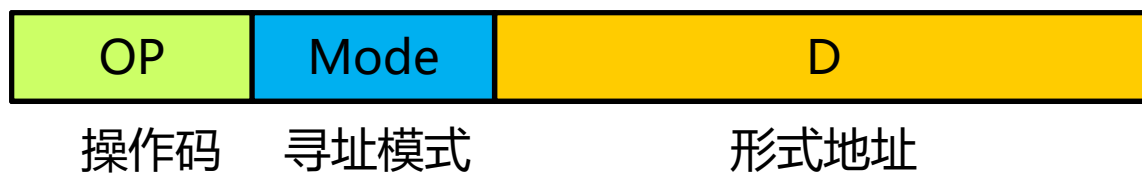


IR(A) → PC

操作数的寻址方式

■ 形成操作数有效地址的方法

- 单地址指令地址码的构成: mode , D
- 实际有效地址为 E, 实际操作数 S
- $S = (E)$

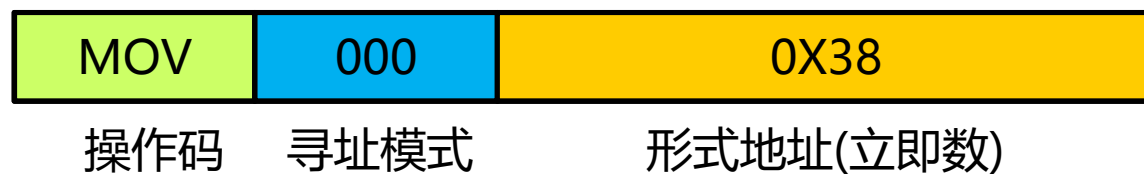


立即寻址

■ 地址码字段是操作数本身

□ S=D

□ 例: MOV AX,38H (38H → AX)

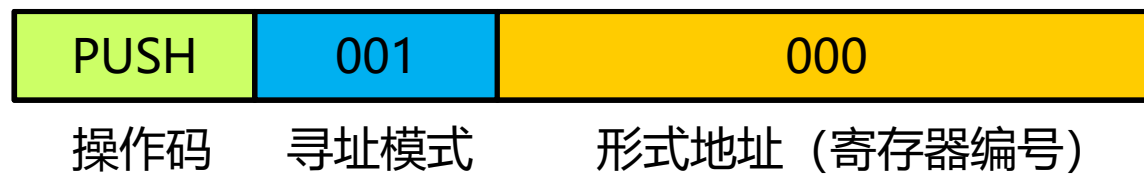


寄存器寻址(Register Addressing)

■ 操作数在CPU的内部寄存器中.

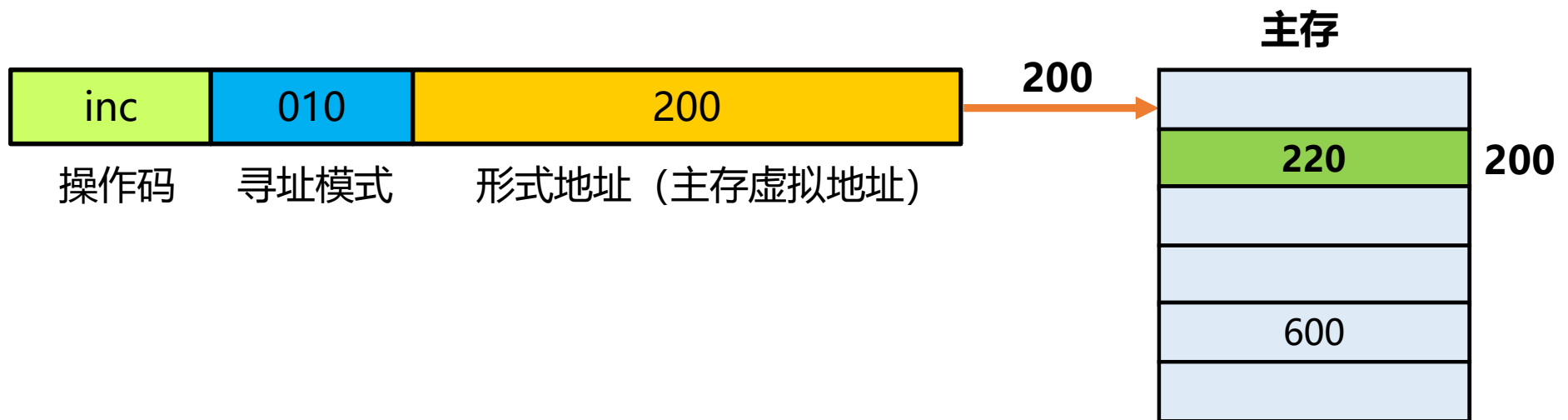
□ AX,BX,CX,DX

□ PUSH AX E=R



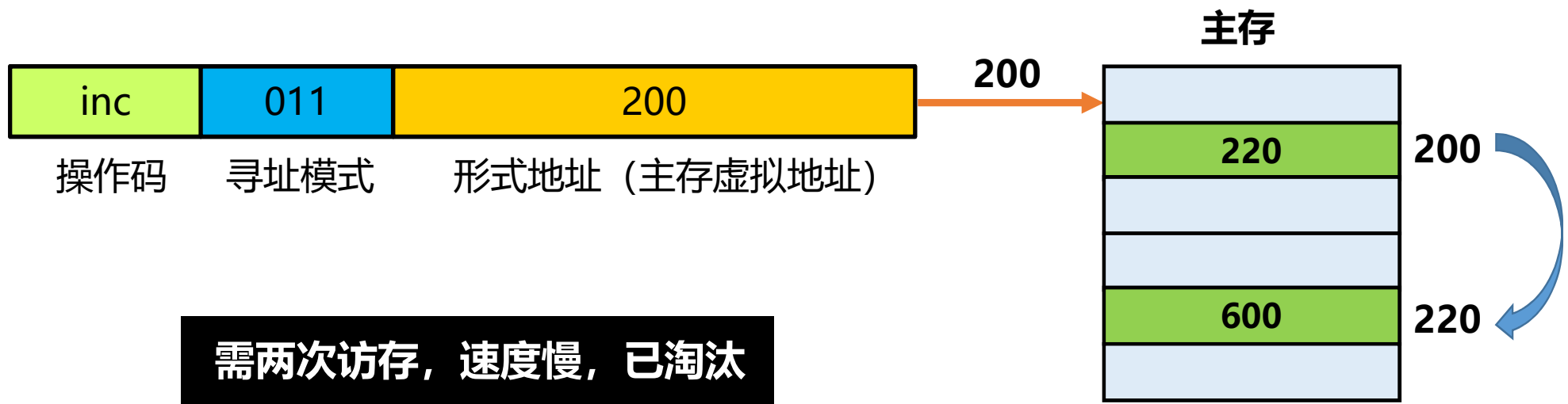
直接寻址(Direct Addressing)

- 地址码字段直接给出操作数在内存的地址. $E=D$
- `inc [200]`



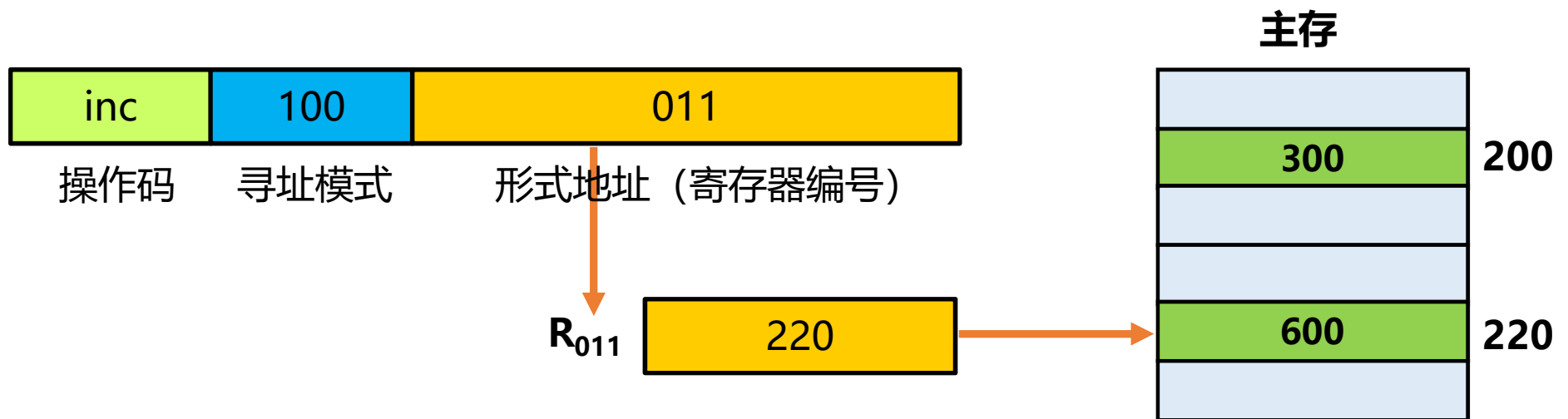
间接寻址(Indirect Addressing)

- D单元的内容是操作数地址, D是操作数地址的地址
- $E = (D)$ $S = ((D))$



寄存器间接寻址 (Register Indirect Addressing)

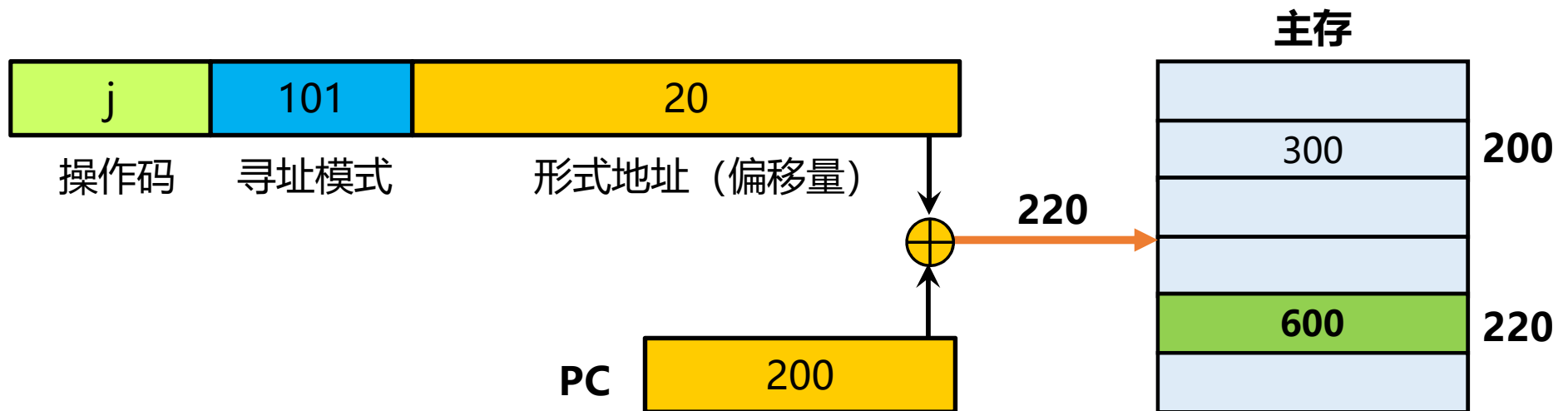
- D单元的内容是操作数的地址,D是操作数地址的地址
- $E=(R)$ `inc [BX]`



相对寻址 (Relative Addressing)

- 指令中的D加上PC的内容作为操作数的地址.
- $E = D + (PC)$

(PC)+D 还是 (PC)+1+D?



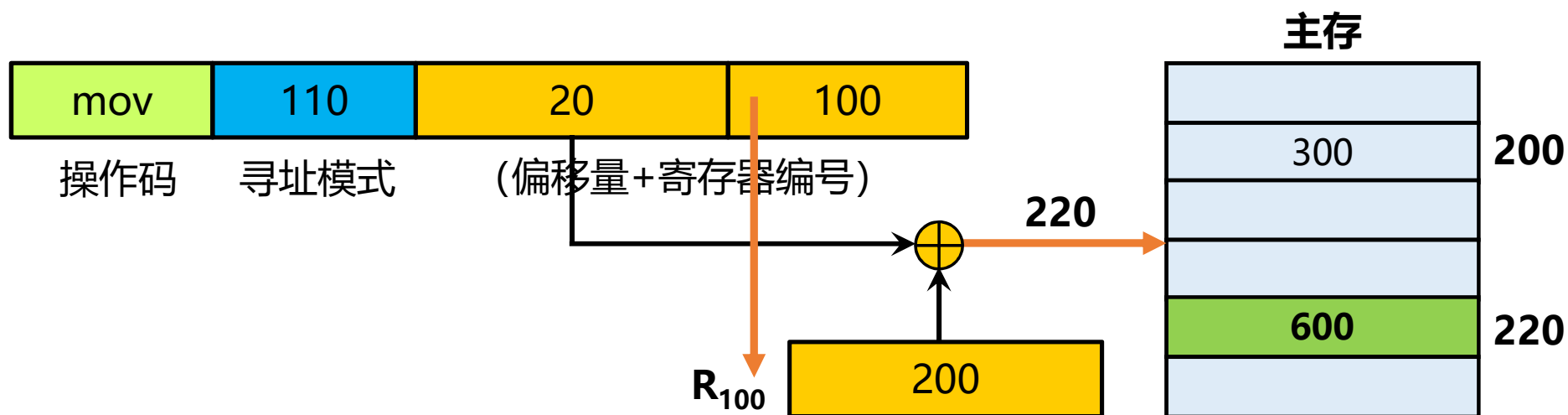
基址/变址寻址

■ 操作数地址为基址/变址寄存器 + 偏移量 基址寄存器一般不修改

■ $E = D + (R)$

■ `MOV AX, 32[SI]`

SI, DI 都称为变址寄存器



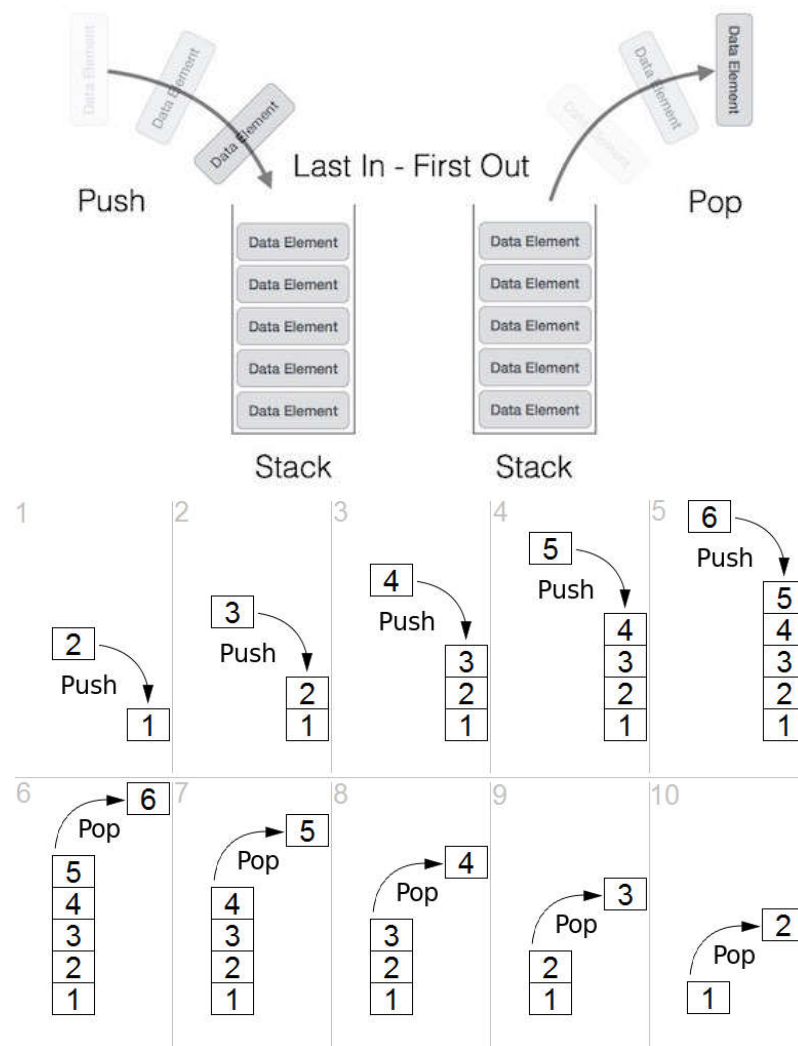
堆栈寻址方式

■ 硬件堆栈（寄存器串联堆栈）

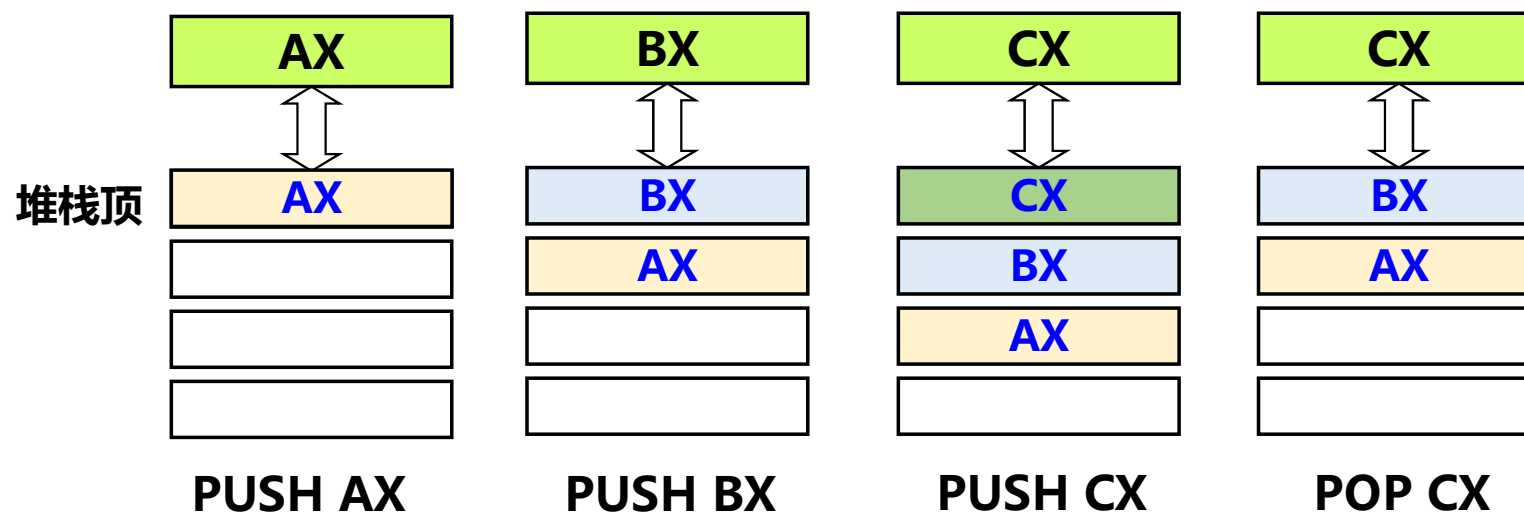
- CPU内部一组串联的寄存器
- 数据的传送在栈顶和通用寄存器之间进行
- 栈顶不动，数据移动，进出栈所有数据都需移动
- 栈容量有限

■ 软件堆栈（内存堆栈）

- 内存区间做堆栈
- SP---堆栈指示器(栈指针),改变SP即可移动栈顶位置。
- 栈顶移动，数据不动，非破坏性读出
- 栈容量大，栈数目容量均可自定义



硬件堆栈

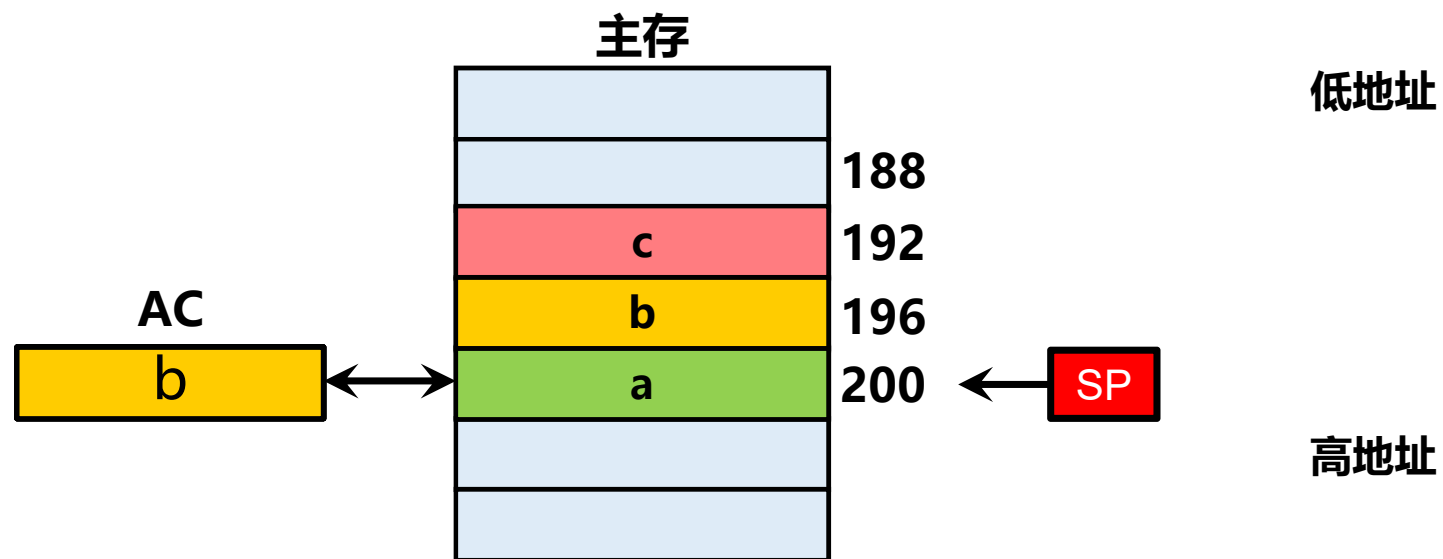


栈顶不动，数据移动

内存堆栈

■ 进栈: $(AC) \rightarrow Mem[sp--]$ 出栈: $Mem[++sp] \rightarrow AC$

■ Push a 、 Push b、 Push c 、 Pop 、 Pop



不同寻址方式对比

不同寻址方式的区别?

	5bits	3bits	8bits		
	操作码	寻址模式	形式地址D	实地址E	寻址范围
立即寻址	MOV	000	38H	S=D	0~255 -128~127
寄存器寻址	MOV	001	00	E=R	0~255# Reg
直接寻址	MOV	010	200	E=D	0~255 RAM Cell
间接寻址	MOV	011	200	E=(D)	0~2 ¹⁶ -1 RAM Cell
寄存器间接	MOV	100	01	E=(R)	0~2 ¹⁶ -1 RAM Cell
相对寻址	JMP	101	20	E=(PC)+D	PC-128~PC+127
变址寻址	MOV	110	20 100	E=(R)+D	0~2 ¹⁶ -1 RAM Cell

寻址方式举例

设某机的指令字长16位，格式、有关寄存器和主存内容如下，X为寻址方式，D为形式地址，请在下表中填入有效地址E及操作数的值。？

OP	X	D=100
----	---	-------

PC=1000

$R_{基}=2000$

100	200
200	500
500	800
1100	100
1102	350
2100	200








寻址方式	X	有效地址E	操作数
立即	0	$S=D$	100
直接	1	$E=D=100$	200
间接	2	$E=(D)=200$	500
相对	3	$E=(PC)+D=1100$	100
变址	4	$E=(R)+D=2100$	200
变址间址	5	$E=((R)+D)=200$	500

|| 本章主要内容

- 指令系统概述
- 指令格式
- 寻址方式
- **RISC 与 CISC**
- MIPS指令系统
- 其他指令系统

指令集体系结构 Instruction Set Architecture (ISA)

■ 不同类型的CPU执行不同指令集，是设计CPU的依据

- ◆  1970 DEC **PDP-11** 1992 **ALPHA**(64位)
- ◆  1978 **x86**, 2001 **IA64**
- ◆  1980 **PowerPC**
- ◆  1981 **MIPS**
- ◆  1985 **SPARC**
- ◆  1991 **arm**
- ◆  2016 **RISC-V**

■ 指令集优劣

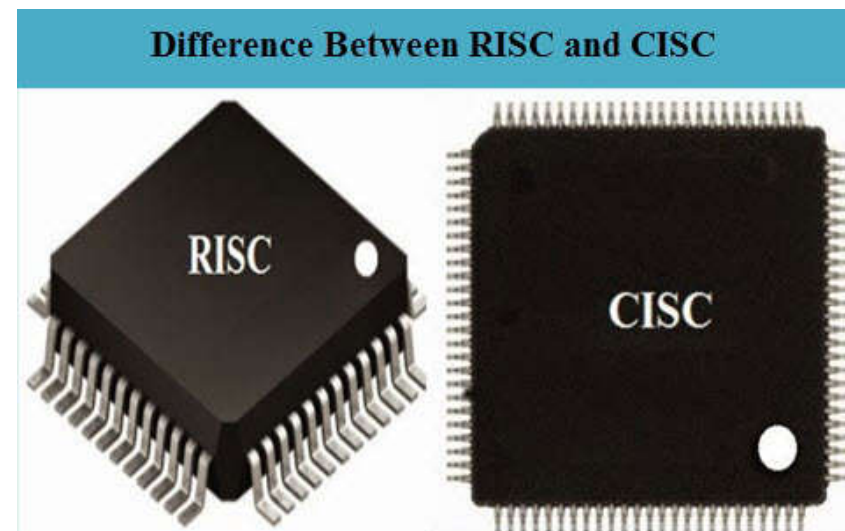
□ 方便硬件设计，方便编译器实现，性能更优，成本功耗更低

硬件设计四原则

- 简单性来自规则性
 - Simplicity favors regularity
 - 指令越规整设计越简单
- 越小越快
 - Smaller is faster
 - 面积小，传播路径小，门延迟少
- 加快经常性事件
 - Make the common case fast
- 好的设计需要适度的折衷
 - Good design demands good compromises

指令系统发展方向

- CISC---复杂指令系统计算机
 - Complex Instruction System Computer
 - 指令数量多，指令功能，复杂的计算机。
 - **Intel X86**
- RISC---精简指令系统计算机
 - Reduced Instruction System Computer
 - 指令数量少，指令功能单一的计算机。
 - 1982年后的指令系统基本都是RISC
 - **MIPS、RISC-V**
- **CISC、RISC互相融合**



精减指令系统(RISC)

- 指令条数少，只保留使用频率最高的简单指令，指令定长
 - 便于硬件实现，用软件实现复杂指令功能
- Load/Store架构
 - 只有存/取数指令才能访问存储器，其余指令的操作都在寄存器之间进行
 - 便于硬件实现
- 指令长度固定，指令格式简单、寻址方式简单
 - 便于硬件实现
- CPU设置大量寄存器（32~192）
 - 便于编译器实现
- 一个机器周期完成一条机器指令
- RISC CPU采用硬布线控制，CISC采用微程序

|| 本章主要内容

- 指令系统概述
- 指令格式
- 寻址方式
- RISC 与 CISC
- **MIPS指令系统**
- 其他指令系统

MIPS指令概述

■ MIPS (Microprocessor without Interlocked Pipeline Stages)

- 1981年斯坦福大学Hennessy教授研究小组研制并商用
- 简单的Load/Store结构
- 易于流水线CPU设计
- 易于编译器开发
- 寻址方式，指令操作非常简单
- MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, **MIPS32**, 和MIPS64多个版本



Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.

- 广泛用于嵌入式系统，在PC机、服务器中也有应用
- 更适合于教学，相比X86更加简洁雅致，不会陷入繁琐的细节

汇编语言的变量---寄存器

- 汇编语言不能使用变量（C、JAVA可以）
 - `int a; float b;`
 - 寄存器变量没有数据类型
- 汇编语言的操作对象是寄存器
 - 好处：寄存器是最快的数据单元
 - 缺陷：寄存器数量有限，需仔细高效的使用各寄存器
- MIPS包括32个通用寄存器，字长---32bits= 1 Word
 - \$0, \$1, \$2, ... \$30, \$31

32个MIPS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)

- 32个32位通用寄存器\$0~\$31
- 32个32位单精度浮点寄存器f₀-f₃₁
- 2个32位乘、商寄存器 H_i 和L₀
- 程序寄存器PC是单独的寄存器
- 无程序状态寄存器
- RISC-V也有类似的32个寄存器设置

IA-32的寄存器组织

%eax	累加器 (32bits)	%ax (16bits)	%ah (8bits)	%al (8bits)
%ecx	计数寄存器	%cx	%ch	%cl
%edx	数据寄存器	%dx	%dh	%dl
%ebx	基址寄存器	%bx	%bh	%bl
%esi	源变址寄存器	%si		
%edi	目标变址寄存器	%di		
%esp	堆栈指针	%sp		
%ebp	基址指针	%bp		
%eip	指令指针	ip		
%eeflags	标志寄存器	flags		

- 8个通用寄存器
- 两个专用寄存器
- 6个段寄存器

CS (代码段) 16bits
SS (堆栈段)
DS (数据段)
ES (附加段)
FS (附加段)
GS (附加段)

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

历史机型中的寄存器数目

年代	机型	通用寄存器个数	体系结构类型	指令形式
1949	EDSAC	1	累加器	ADD 200 (AC)+200→AC,
1953	IBM 701	1	累加器	
1963	CDC 6600	8	Load-Store (Register-Register)	
1964	IBM S/360	16	Register-Memory	一个操作数在内存中, 一个在寄存器
1965	DEC PDP-8	1	累加器	
1970	DEC PDP-11	8	Register-Memory	
1972	Intel 8008	1	累加器 (1个累加器+6个通用寄存器+2个ALU暂存寄存器)	
1974	Motorola 6800	2	累加器	
1977	DEC VAX	16	Register-Memory, Memory-Memory	两个操作数可同时在内存中
1978	Intel 8086	1	扩展的累加器/专用寄存器	
1980	Motorola 68000	16	Register-Memory	
1985	Intel 80386	8	Register-Memory	
1985	MIPS	32	Load-Store	
1986	HP PA-RISC	32	Load-Store	
1987	SUN SPARC	32	Load-Store	
1992	IBM PowerPC	32	Load-Store	
1992	DEC Alpha	32	Load-Store	

MIPS指令分类

■ 运算指令

- 算术: add,addi,addu,addiu, sub,subu, mult,multu,div,divu, slt,slti,sltiu
- 逻辑: and,andi,or, ori,xor,xori,nor
- 移位指令: sll,sllv,srl, srlv,sra,srav

■ 分支指令

- beq, bne, blez(≤ 0), bgez(≥ 0), bltz(< 0), bgtz(> 0), jal, j, jr

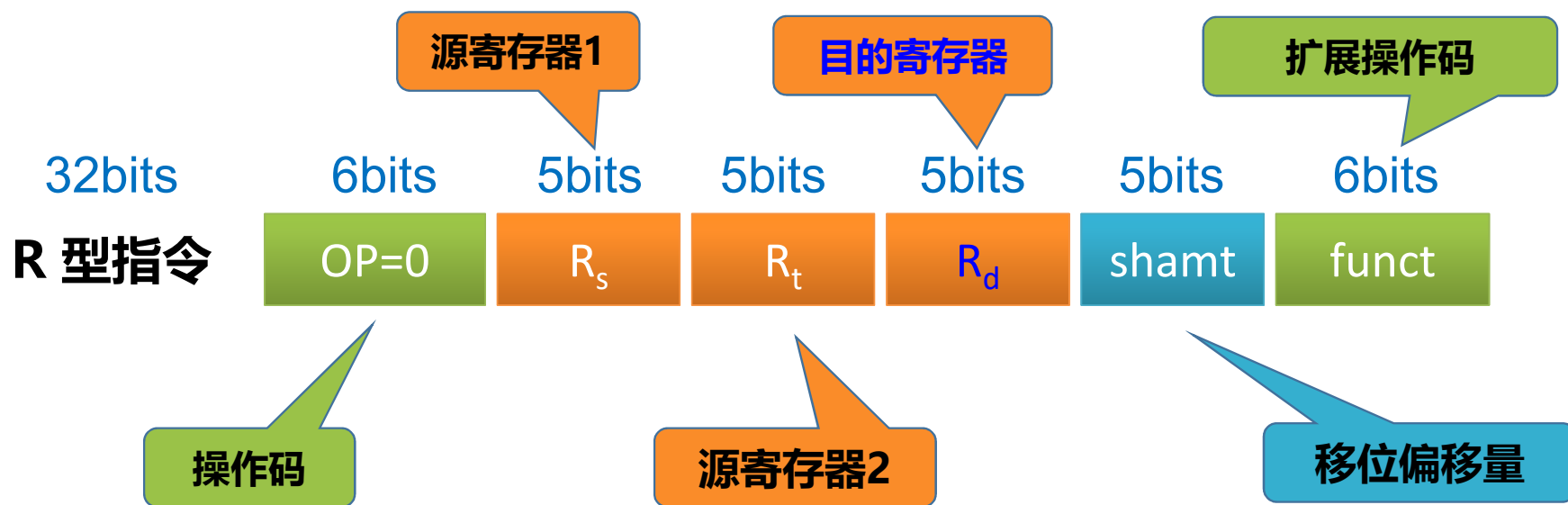
■ 访存指令

- lw,lh,lb,sw,sh,sb

■ 系统指令

- syscall, break, sync, cache

MIPS 32指令格式 (R型指令)



无寻址方式字段，隐藏在操作码字段OP中

MIPS指令格式



MIPS指令格式 (R型指令)

指令	格式	6bits OP	5bits rs	5bits rt	5bits rd	5bits shamt	6bits funct
add	R	0	Reg	Reg	Reg	0	32 ₁₀
sub	R	0	Reg	Reg	Reg	0	34 ₁₀
and	R	0	Reg	Reg	Reg	0	36 ₁₀
or	R	0	Reg	Reg	Reg	0	37 ₁₀
nor	R	0	Reg	Reg	Reg	0	39 ₁₀
sll	R	0	0	Reg	Reg	X	0 ₁₀
srl	R	0	0	Reg	Reg	X	2 ₁₀
jr	R	0	Reg			0	8 ₁₀

add	R	0	18	19	17	0	32
-----	---	---	----	----	----	---	----

■ add \$s1, \$s2, \$s3

左侧为目的操作数

机器码 0x2538820

MIPS指令格式 (I、J型指令)

		6bits	5bits	5bits	5bits	5bits	6bits
指令	格式	OP	rs	rt	rd	shamt	funct
add	R	0	Reg	Reg	Reg	0	32 ₁₀
addi	I	8	Reg	Reg	16bits 立即数		
lw	I	35	Reg	Reg	16bits 立即数		
sw	I	43	Reg	Reg	16bits 立即数		
andi	I	12	Reg	Reg	16bits 立即数		
ori	I	13	Reg	Reg	16bits 立即数		
beq	I	4	Reg	Reg	16bits 立即数 (相对寻址)		
bne	I	5	Reg	Reg	16bits 立即数 (相对寻址)		
j	J	2	26bits 立即数(伪直接寻址)				
jal	J	3	26bits 立即数(伪直接寻址)				

MIPS寻址方式

- 寄存器寻址：操作数为寄存器
- 变址寻址：寄存器的值+偏移量 作为存储单元的地址
- 立即数寻址
- PC相对寻址 `beq reg1, reg2, offset`
 - $PC + 4 + 16\text{位偏移地址左移两位}$
- 伪直接寻址 `J label`



|| 加减指令

■ 加法

□ $a = b + c$ (in C)

□ `add $s0, $s1, $s2` (in MIPS)

□ a, b, c 编译后对应寄存器 `$s0, $s1, $s2`

■ 减法

□ $d = e - f$ (in C)

□ `sub $s3, $s4, $s5` (in MIPS)

□ d, e, f 编译后对应寄存器 `$s3, $s4, $s5`

|| 加减指令

- 如何编译下面的C语言表达式?

$a = b + c + d - e;$

- 编译成多行汇编指令

`add $t0, $s1, $s2 # temp = b + c`

`add $t0, $t0, $s3 # temp = temp + d`

`sub $s0, $t0, $s4 # a = temp - e`

- 一个简单的C语言表达式变成多行汇编语句

内存数据访问指令 lw sw lb sb lh sh

■ 读内存指令

□ `g = h + A[8];` (in C)

□ `lw $t0, 32($s3)` # `$s3`为A[0]地址 (in MIPS)

□ `add $s1, $s2, $t0` # `g=h+A[8]`

■ 变址寻址

□ 基址寄存器 + 偏移量

■ 写内存指令

□ `A[12] = h + A[8];`

□ `lw $t0, 32($s3)` # `get A[8]` (in MIPS)

□ `add $t0, $s2, $t0` # `h+A[8]`

□ `sw $t0, 48($s3)` # `store A[12]`

加立即数

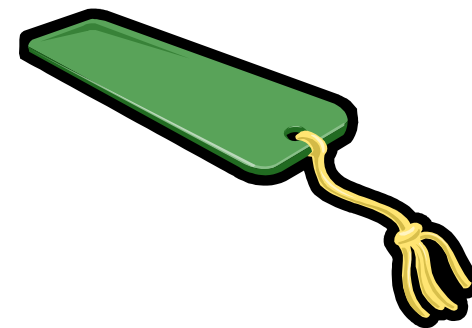
■ 常数相加指令

- `g = g + 4;` (in C)
- `lw $t0, 0($s3) # $t0=4 $s3=Address(4)`
- `add $s1,$s1,$t0 # g=g+4`

■ 立即数相加指令

- `addi $s1,$s1,4 # $s1=$s1+4` (in MIPS)

指令	实例	语义	注释
加	<code>add \$s1,\$s2,\$s3</code>	$s1 = s2 + s3$	寄存器寻址
减	<code>sub \$s1,\$s2,\$s3</code>	$s1 = s2 - s3$	寄存器寻址
加立即数	<code>addi \$s1,\$s2,100</code>	$s1 = s2 + 100$	寄存器寻址+立即数寻址
取字	<code>lw \$s1,100(\$s2)</code>	$s1 = \text{Mem}[s2 + 100]$	寄存器寻址+变址寻址
存字	<code>sw \$s1,100(\$s2)</code>	$\text{Mem}[s2 + 100] = s1$	寄存器寻址+变址寻址



条件判断指令 `beq reg1, reg2, label`

■ C语言条件判断指令

```
if (a==b)
{ i=1; }
else
{ i=2; }
```

■ 等效C指令

```
if (a==b) goto L1;
i=2;
goto L2;
L1: i=1;
L2:
```



■ MIPS数据传送指令

```
addi $s3,$zero,1
# $s3=1      立即数传送
add $s3,$s2,$zero
# $s3=$s2    寄存器传送
```

■ 等效MIPS指令

```
beq $s0,$s1,L1
addi $s3,$zero,2
j L2;
L1: addi $s3,$zero,1
L2:
```

|| MIPS 条件判断指令

■ 条件跳转

□ `If (reg1==reg2) goto Label1` (C语言)

□ `beq reg1,reg2,Label1` (MIPS指令)

□ `bne reg1,reg2,Label2`

■ 无条件跳转指令

□ `goto Label;` (C语言)

□ `j label` (MIPS指令)

□ `beq $zero,$zero,label` (MIPS指令)

◆ 不能完全等效?

相对寻址, label可正可负

|| If-else语句举例 X86机器级表示

```
3      {  
0x00401334    push    %ebp  
0x00401335    mov     %esp,%ebp  
0x00401337    and     $0xffffffff0,%esp  
0x0040133A    sub     $0x10,%esp  
0x0040133D    call    0x401910 <__main>  
4          int i,result;  
5          if (i)  
0x00401342    cmpl    $0x0,0xc(%esp)  
0x00401347    je      0x401353 <main+31>  
6          result=0;  
0x00401349    movl    $0x0,0x8(%esp)  
0x00401351    jmp     0x40135b <main+39>  
7          else result=1;  
0x00401353    movl    $0x1,0x8(%esp)  
8      }  
0x0040135B    leave  
0x0040135C    ret
```

```
#include <stdio.h>  
int main ()  
{  
    int i,result;  
    if (i)  
        result=0;  
    else result=1;  
}
```

逻辑运算

■ 移位指令

□ `a=b<<2;` C语言

□ `sll,srl,sra` `sll $s1,$s2,2` `#s1=s2<<2`

□ `sllv,srlv,srav` `sllv $s1,$s2,$s3` `#s1=s2<<s3`

移位偏移量最多5位

■ 逻辑运算

□ `and,or,xor,nor` `and $t0,$t1,$t2` `#t0=t1&t2`

□ `andi,ori,xori` `and $t0,$t1,100` `# t0=t1&100`

循环结构

■ C语言简单循环结构，A为int数组

```
do {  
    g = g + A[i];  
    i = i + j;  
} while (i != h);
```

■ 重写代码

```
Loop: g = g + A[i];  
      i = i + j;  
      if (i != h) goto Loop;
```

■ 编译后的变量映射:

g	h	i	j	A[0]
\$s1	\$s2	\$s3	\$s4	\$s5

循环结构

■ 最后编译的MIPS代码:

```
Loop: sll    $t1, $s3, 2          # $t1= 4*i
      addu   $t1, $t1, $s5        # $t1=&A[0]+4i
      lw     $t1, 0($t1)         # $t1=A[i]
      addu   $s1, $s1, $t1       # g=g+A[i]
      addu   $s3, $s3, $s4       # i=i+j
      bne    $s3, $s2, Loop      # if i!=h goto Loop
```

■ 原始C代码: `Loop: g = g + A[i];`
 `i = i + j;`
 `if (i != h) goto Loop;`

do-while语句举例

```
3      {  
0x00401334    push    %ebp  
0x00401335    mov     %esp,%ebp  
0x00401337    and     $0xffffffff0,%esp  
0x0040133A    sub     $0x10,%esp  
0x0040133D    call    0x401910 <__main>  
4          int i=0;  
0x00401342    movl    $0x0,0xc(%esp)  
5          do  
6          {  
7              i++;  
0x0040134A    incl    0xc(%esp)  
8          }while(i>0);  
0x0040134E    cmpl    $0x0,0xc(%esp)  
0x00401353    jg      0x40134a <main+22>  
9          }  
0x00401355    leave  
0x00401356    ret
```

```
#include <stdio.h>  
int main ( )  
{  
    int i=0;  
    do  
    {  
        i++;  
    }while(i>0);  
}
```

比较指令 `slt` `slti`

■ MIPS比较指令 (Set on Less Than)

■ `slt reg1, reg2, reg3`

`reg1 = (reg2 < reg3) ? 1 : 0;` (C语言)

利用通用寄存器存储比较结果!

■ `If (g<h) goto Less;`

`slt $t0, $s0, $s1 # $t0 = 1 if g<h`

`bne $t0, $0, Less # if $t0!=0 goto Less:`

`blt` 伪指令 (branch less than)

|| MIPS过程调用

■ C语言函数调用

```
int function(int a ,int b)  
{ return (a+b); }
```

■ MIPS实现过程调用的机制

- 返回地址寄存器 **\$ra**
- 参数寄存器 **\$a0, \$a1, \$a2, \$a3**
- 返回值寄存器 **\$v0 \$v1**
- 局部变量 **\$t0~\$t9**
- 堆栈指针 **\$sp**

过程调用实现机制

```
sum(a,b); /* a,b:$s0,$s1 */
}
int sum(int x, int y)
{ return x+y; }
```

```
1000 add  $a0,$s0,$zero    # x = a      传参
1004 add  $a1,$s1,$zero    # y = b      传参
1008 addi $ra,$zero,1016    # $ra=1016  保存返回地址
1012 j     sum             # 跳转, 调用过程sum
1016
```

```
1008 jal sum
1012
```

...

```
2000 sum: add $v0,$a0,$a1  # 过程入口
2004 jr     $ra            # 返回主程序
```

```
J 1016
```

过程调用机制

```
jal Label    # jump and link
```

■ 等效于如下指令

```
$ra=PC+4; #save next instruction address
```

```
j Label
```

■ 过程返回指令

```
jr $ra      #return to main program
```

■ 问题：利用\$ra做返回地址，如果过程嵌套如何返回？

- \$ra 会被多次覆盖

- 利用堆栈保存\$ra

多级过程调用

```
int sumSquare(int x, int y)

{ return mult(x,x)+ y; }
```

- 主程序调用`sumSquare(x, y)`时 `$ra`保存一次，保证该过程执行完毕后能返回主程序。
- 调用 `mult`时会覆盖`$ra`
 - 需要保存 `sumSquare`的返回地址
- 其它被复用的寄存器`$a0, $a1`也存在同样的问题，这是寄存器传参的弊端

堆栈操作

■ sumSquare:

	<code>addi \$sp,\$sp,-8</code>	<code># space on stack</code>
"push"	<code>sw \$ra, 4(\$sp)</code>	<code># save ret addr</code>
	<code>sw \$a1, 0(\$sp)</code>	<code># save y</code>
	<code>add \$a1,\$a0,\$zero</code>	<code># mult(x,x)</code>
	<code>jal mult</code>	<code># call mult</code>
"pop"	<code>lw \$a1, 0(\$sp)</code>	<code># restore y</code>
	<code>add \$v0,\$v0,\$a1</code>	<code># mult()+y</code>
	<code>lw \$ra, 4(\$sp)</code>	<code># get ret addr</code>
	<code>addi \$sp,\$sp,8</code>	<code># restore stack</code>
	<code>jr \$ra</code>	

■ mult: ...

- 注意：除了返回地址以外，函数参数等会覆盖的变量都需要入栈

函数调用的机器级表示

- 调用子程序包含两个参与者
 - 调用者 (caller)
 - ◆ 准备函数参数，跳转到被调用者子程序
 - 被调用者 (callee)
 - ◆ 使用调用者提供的参数，然后运行
 - ◆ 运行结束保存返回值
 - ◆ 将控制（如跳回）还给调用者。

函数调用的机器级表示

- 高级语言函数体中一般使用局部变量
- 汇编子程序使用寄存器（全局变量）
- 对全局变量的修改可能会引起调用者逻辑不正确
- 调用者函数和被调用函数可能使用相同寄存器
 - 造成数据破坏？
 - 被调用函数需要保存可能被破坏的寄存器（现场）
 - 哪些寄存器属于现场？

ISA寄存器使用约定

■ 调用者保存寄存器

- 调用者负责根据需要保存现场
- 被调用过程可直接使用，不用压栈。

■ **X86:** IA32 EAX、EDX、ECX **MIPS:** \$t0~\$t9

■ 被调用者保存寄存器

- 被调用者负责根据需要保存现场 save
- 返回之前恢复它们的值 restore

■ **X86:** IA32 EBX、ESI、EDI **MIPS:** \$s0~\$s7, \$fp,\$ra

■ 为减少开销，每个过程应优先使用哪些寄存器？

Intel 函数参数传递

■ X86参数传递

- 栈帧

■ Linux IA-64参数传递

- 先rdi,rsi,rdx,rcx,r8和r9, 浮点数xmm0-xmm7
- 剩余的由右向左依次入栈

■ Windows IA-64参数传递

- rcx,rdx,r8,r9, 浮点数xmm0-xmm3
- 剩余由右向左依次入栈

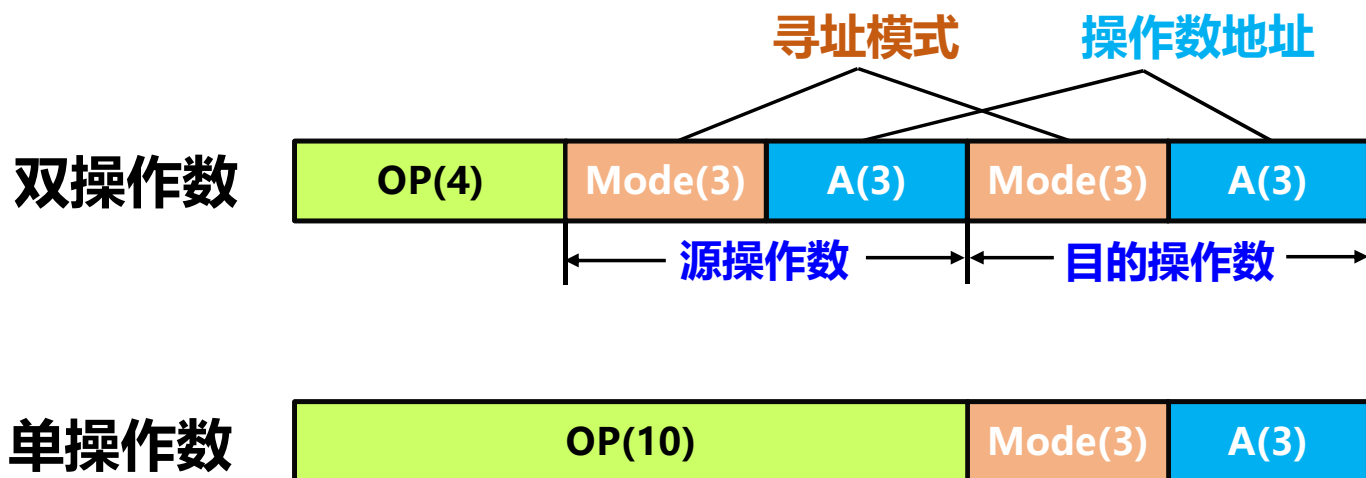
ABI (application binary interface)

- 描述应用程序和操作系统之间，应用和库之间，或应用各部分间的接口
 - 数据类型的大小、布局和对齐
 - 调用约定（控制函数参数如何传送以及如何接受返回值）
 - ◆ 所有参数都通过栈传递，还是部分参数通过寄存器传递
 - ◆ 哪个寄存器用于哪个函数参数
 - ◆ 栈传递的第一个函数参数是最先push到栈上还是最后
 - 系统调用的编码和一个应用如何向操作系统进行系统调用
 - 目标文件的二进制格式、程序库等等

|| 本章主要内容

- 指令系统概述
- 指令格式
- 寻址方式
- RISC 与 CISC
- MIP指令系统
- **其它指令系统**

指令格式举例 PDP-11



- 1957年DEC公司成立，生产小型计算机
- 1970年PDP-11诞生
 - 70~80年代红极一时，后被苹果II，IBM-PC超越
- 1984年VAX8600 扳回一局
- 1998年被Compaq 96亿美金收购，2002并入惠普



PDP-11指令集特点



- 机器字长16位
- 单字长，双字长，三字长指令
 - 单字长后续两个字可用于变址偏移量，内存地址，立即数，最多3字长
- 8种寻址方式
- 8个16位寄存器r0~r7，有条件状态寄存器PSW
 - r0~r5通用寄存器，r6为栈指针SP，r7为程序计数器PC
- 较好的规整性，典型的扩展操作码指令

digital

PDP-11寻址方式

mode	寻址方式	汇编语法	功能
0	寄存器	R_i	寄存器值就是操作数
1	寄存器 间接	(R_i)	寄存器的值是操作数地址
2	自增寻址	$(R_i) +$	寄存器的值是操作数地址，取数后寄存器自增 (byte +1, word +2)
3	自增 间接	$@(R_i) +$	寄存器的值是操作数地址的地址，取数后寄存器加2
4	自减寻址	$-(R_i)$	先将寄存器自减，运算结果是操作数地址 (byte -1, word -2)
5	自减 间接	$@-(R_i)$	先将寄存器减2，运算结果是操作数地址的地址
6	变址寻址	$\text{index}(R_i)$	操作数地址 = 寄存器的值 + 16位index
7	变址 间址	$@\text{index}(R_i)$	操作数地址的地址 = 寄存器的值 + 16位index

C语言风格, 适合堆栈指令

■ **MOV R1, (R0)** **MOV (R0) +, -(SP)** **ADD @1000(R2), 200(R1)**

X86指令格式 (最长15bytes)

■ JE 20H

4bits	4bits	8bits
JE 4	Cond.	Displacement

■ Call

8bits	32bits
Call	Offset

■ PUSH ESI

5bits	3bits
PUSH	Reg

■ MOV EBX,[EDI+45]

6bits	1	1	8Bits	8Bits
MOV	d	w	Postbyte	Displacement

MIPS X86 差异

#	X86	MIPS
1	变长 (1-15bytes)	定长指令
2	指令数多 CISC	指令数少 RISC
3	8个通用寄存器	32个通用寄存器
4	寻址方式复杂	寻址方式简单
5	有标志寄存器	无标志寄存器
6	最多两地址指令	三地址指令
7	无限制	只有Load/store能访问存储器
8	有堆栈指令 push, pop	无堆栈指令 (访存指令代替)
9	有I/O指令	无I/O指令(设备统一编址)
10	参数传递: 栈帧	参数传递 (4寄存器+栈帧)

MIPS 32 & ARMv8-32

arm

■ 相同之处

- 32位定长指令
- 32个通用寄存器，一个恒零寄存器
- load/store架构
- 都不能并行存取多个寄存器（方便批量保存寄存器，恢复寄存器，硬件实现更复杂）
- 都有分支指令，能根据寄存器的值为0转移或不为零转移

■ 区别

- 条件分支指令，arm依赖于条件码，mips无状态标志寄存器
- ARMv7指令还有**条件执行指令**，不满足条件不执行
- ARMv8指令集规模更大，寻址方式更多

RISC-V

■ 完全开放的 ISA

■ 大道至简，简单就是美

- 包含一个最小的**核心冻结的ISA**（可支撑OS，方便教学）
- 适合硬件实现，而不仅仅是适用于模拟或者二进制翻译

■ 无病一生轻的后发优势

- 模块化的可扩展指令集
- 方便简化硬件实现，提升性能
 - ◆ 更规整的指令编码、更简洁的运算指令、更简洁的访存模式：Load/Store架构
 - ◆ 高效分支跳转指令（减少指令数目）、简洁的子程序调用
 - ◆ 无条件码执行、无分支延迟槽



MIPS 32 & RISC-V

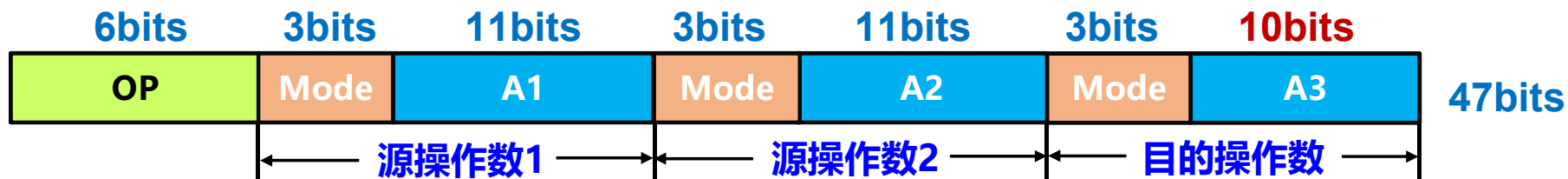
- 指令助记符及语法格式大同小异
- RISC-V 分支预测，MIPS延迟槽
- RISC-V支持变长指令扩展
- RISC-V 将源寄存器rs1，rs2和目标寄存器（rd）固定在同样位置，以简化指令译码
- 立即数分散在不同位置，但符号位固定在第31位，可加速符号扩展电路，与译码并行



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

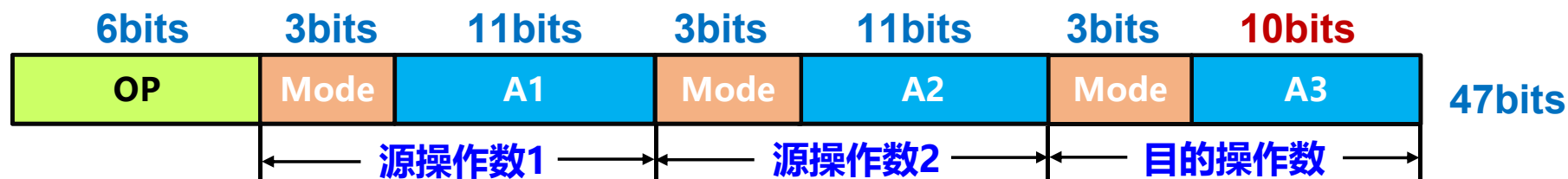
指令格式设计举例

- **例2.** 某机字长32位，按32位编址，采用三地址指令，支持8种寻址操作，完成60种操作，各寻址方式均可在2K主存范围内取得操作数，并可在1K范围内保存运算结果。问应采用什么样的指令格式？指令字长最少应为多少位？执行一条直接寻址模式指令最多要访问多少次主存？
- 三操作数，每个操作数包括寻址方式和寻址范围，寻址方式是 3 b，源操作数地址字段是 11 b，目的操作地址字段是 10 b
- 操作码需要 6 b



指令格式设计举例

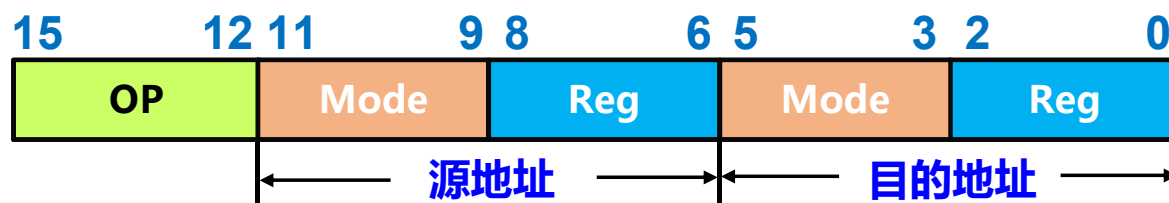
- **例2.** 某机字长32位，采用三地址指令，支持8种寻址操作，完成60种操作，各寻址方式均可在2K主存范围内取得操作数，并可在1K范围内保存运算结果。问应采用什么样的指令格式？指令字长最少应为多少位？执行一条直接寻址模式指令最多要访问多少次主存？



- 47位指令字需占用2个存储字，取指需访存2次
- 取源操作数访存2次，写结果1次，共5次

指令格式设计举例

■ 例3. 分析以下指令格式及寻址方式特点？



- 1) ____地址指令？
 - 2) 操作码可指定____条指令？
 - 3) 源和目的均有____种寻址方式？
 - 4) 源地址寄存器和目的地址寄存器均有____个；
 - 5) 可寻址范围为____K
- 1) 2地址指令；
 - 2) 操作码可指定16条指令；
 - 3) 源和目的均有8种寻址方式；
 - 4) 均有8个；
 - 5) 可寻址范围为1~64K（与机器字长有关）

作业

- 5.2
- 5.4
- 5.5
- 5.6
- 5.9
- 5.12



THANKS

计算机组成原理