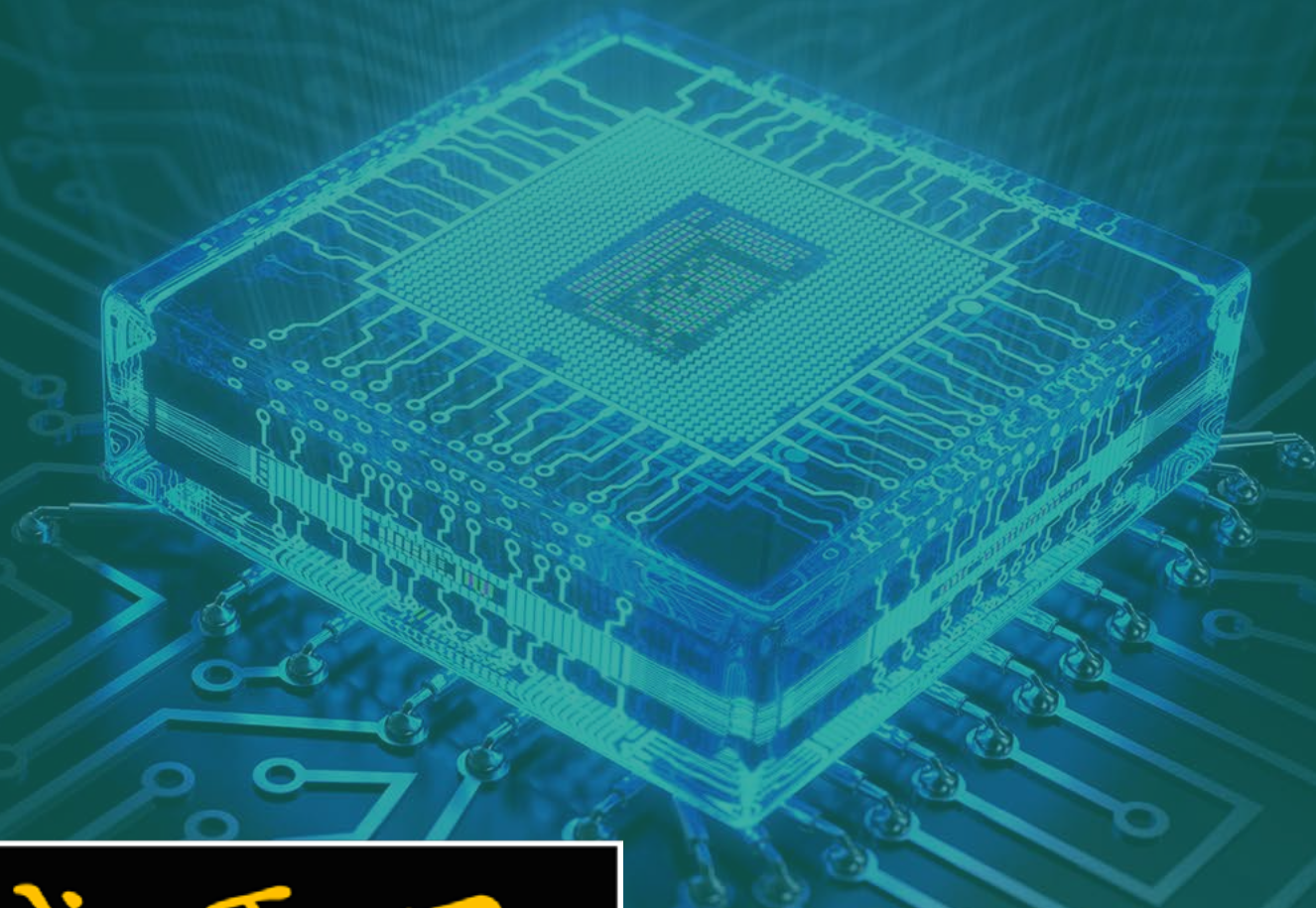




华中科技大学

计算机科学与技术学院

School of Computer Science & Technology, HUST



计算机组成原理



计算机组成原理



附：MIPS 指令系统




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

n 不同类型的CPU执行不同指令集，是设计CPU的依据


u  1970 DEC PDP-11 1992 ALPHA(64位)

u  1978 x86, 2001 IA64

u  1980 PowerPC

u  1981 MIPS

u  1985 SPARC

u  1991 arm

u  2016 RISC-V

2020
LoongArch



n 指令集设计考量

p 方便编译优化，OS，虚拟机开发，方便硬件实现（高性能、低功耗）

|| 硬件设计四原则

n 简单性来自规则性

- p Simplicity favors regularity
- p 指令越规整设计越简单

n 越小越快

- p Smaller is faster
- p 面积小，传播路径小，门延迟少

n 加快经常性事件

- p Make the common case fast

n 好的设计需要适度的折衷

- p Good design demands good compromises

MIPS指令概述

n MIPS (Microprocessor without Interlocked Pipeline Stages)

p 1981年斯坦福大学Hennessy教授研究小组研制并商用

p 简单的Load/Store结构

p 易于流水线CPU设计

p 易于编译器开发

p 寻址方式，指令操作非常简单

p MIPS I，MIPS II，MIPS III，MIPS IV，MIPS V，MIPS32，和MIPS64多个版本

n 广泛用于嵌入式系统，在PC机、服务器中也有应用

n 更适合于教学，相比X86更加简洁，不会陷入繁琐的细节



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

汇编语言的变量---寄存器

n 汇编语言的操作对象是寄存器和内存单元

p 好处：寄存器是最快的存储单元

p 缺陷：数量有限，需仔细高效使用各寄存器

n 寄存器有无数据类型？

p C语言数据类型：char、int、short、float

p 寄存器变量数据类型？

n MIPS寄存器，字长---32bits= 1 Word

n 32个通用寄存器：\$0 ~ \$31

n 32个浮点寄存器：\$f0~\$f31 存放32个单精度或16个双精度浮点数

n 特殊寄存器：Hi, Lo 存放乘除法结果



32个MIPS寄存器 (5位地址)

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，调用者按需保存
16~23	\$s0~\$s7	save寄存器，被调用者按需保存
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)

中断异常时寄存器如何保存?

- n 32个32位通用寄存器\$0~\$31
- n 32个32位单精度浮点寄存器f₀-f₃₁
- n 2个32位乘、商寄存器 H_i 和L₀
- n 程序寄存器PC是单独的寄存器
- n 无程序状态寄存器
- n 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		

- n 8个通用寄存器 (3位地址)
- n 两个专用寄存器
- n 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

p 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)+200à AC ,
1953	IBM 701	1	累加器	
1963	CDC 6600	8	Load-Store (Register-Register)	只有访存指令才能访问内存
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加减指令

n 加法

p $a = b + c$ (in C)

p `add $s0,$s1,$s2` (in MIPS)

p a, b, c 编译后对应寄存器 `$s0,$s1,$s2`

n 减法

p $d = e - f$ (in C)

p `sub $s3,$s4,$s5` (in MIPS)

p d, e, f 编译后对应寄存器 `$s3,$s4,$s5`

|| MIPS加减指令

n 如何编译下面的C语言表达式?

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

n 编译成多条汇编指令

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

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

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

n 简单的C语言表达式变成多条汇编指令

p 指令条数和运算符数目相关

|| MIPS访存指令 lw sw lb sb lh sh

n 读内存指令

p `g = h + A[8];` # A为int数组 (in C)

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

p `add $s1, $s2, $t0` # `g=h+A[8]`

n 变址寻址

p 基址寄存器 + 偏移量

n 写内存指令

p `A[12] = h + A[8];` # A为int数组 (in C)

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

p `add $t0, $s2, $t0` # `h+A[8]`

p `sw $t0, 48($s3)` # `store A[12]`

|| 加立即数

n 加常数运算

p $g = g + 4;$ (in C)

p `lw $t0, 0($s3) # $t0=4 $s3=Address(4)`

p `add $s1,$s1,$t0 # g=g+4`

n 立即数加指令

p `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

|| 条件分支指令 `beq reg1, reg2, label`

n C语言条件判断与分支

```
if (a==b)
```

```
{ i=1; }
```

```
else
```

```
{ i=2; }
```

n 等效C指令

```
if (a==b) goto L1;
```

```
i=2;
```

```
goto L2;
```

```
L1:i=1;
```

```
L2:
```



n 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
```

n 等效MIPS指令

```
beq $s0,$s1,L1
```

```
addi $s3,$zero,2
```

```
j L2;
```

```
L1:addi $s3,$zero,1
```

```
L2:
```

MIPS 条件分支指令

n 条件分支

p if (reg₁==reg₂) goto Label₁ (C语言)

p beq reg₁,reg₂,Label₁ (MIPS指令)

p bne reg₁,reg₂,Label₂

n 无条件分支指令

p goto Label ; (C语言)

p j label (MIPS指令)

p beq \$zero,\$zero,label (MIPS指令)

相对寻址，label可正可负

u 不能完全等效？

|| 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

n 移位指令

p `a=b<<2;` C语言

移位偏移量最多5位

p `sll,srl,sra`

`sll $s1,$s2,2`

`#s1=s2<<2`

p `sllv,srlv,srav`

`sllv $s1,$s2,$s3`

`#s1=s2<<s3`

p 有没有算术左移?

n 逻辑运算

p `and,or,xor,nor`

`and $t0,$t1,$t2`

`#t0=t1&t2`

p `andi,ori,xori`

`and $t0,$t1,100`

`#t0=t1&100`

循环结构

n C语言简单循环结构，A为int数组

```
p do {  
    g = g + A[i];    // 可变索引数组元素访问  
    i = i + j;  
} while (i != h);
```

n 重写代码

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

n 编译后的变量映射:

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

循环结构

n 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
```

n 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);
}
```

|| 比较置位指令 slt slti

n 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！

n blt 伪指令 (branch less than)

```
if (g < h) goto Less
```

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

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

为什么是伪指令？

|| MIPS过程调用

n C语言函数调用

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

n MIPS过程调用机制

p 返回地址寄存器 \$ra

p 参数寄存器 \$a0, \$a1, \$a2, \$a3

p 返回值寄存器 \$v0 \$v1

p 局部变量 \$t0~\$t9 全局变量 \$s0~\$s7

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

```
1008 jal sum
1012
```

```
J 1016
```

过程调用机制

n 过程调用指令

`jal Label # jump and link`

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

`U j Label`

n 过程返回指令

`jr $ra #return to main program`

n 如何实现过程嵌套

p `$ra` 会被多次覆盖

p 利用堆栈保存 `$ra`

|| 多级过程调用 ((嵌套))

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

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

- n 主程序调用sumSquare(x , y)时 \$ra保存一次
 - p 方便返回调用位置
- n 调用 mult 时再保存一次\$ra , 导致\$ra覆盖
- n 其它被复用的寄存器 \$a0,\$a1 也存在同样的问题
- n 寄存器传参的弊端



堆栈操作

n `sumSquare:`

```
addi $sp,$sp,-8      # space on stack
```

"push"

```
sw $ra, 4($sp)        # save ret addr
```

被调用者保存

```
sw $a1, 0($sp)        # save y
```

调用者保存

```
add $a1,$a0,$zero     # mult(x,x)
```

```
jal mult              # call mult
```

"pop"

```
lw $a1, 0($sp)        # restore y
```

```
add $v0,$v0,$a1       # mult()+y
```

```
lw $ra, 4($sp)        # get ret addr
```

```
addi $sp,$sp,8        # restore stack
```

```
jr $ra
```

n `mult: ...`

n 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

n 调用子程序包含两个参与者

p 调用者 (caller)

- u 准备函数参数
- u 保存返回地址
- u 跳转到被调用者子程序

p 被调用者 (callee)

- u 使用调用者提供的参数，然后运行
- u 运行结束保存返回值
- u 将控制（如跳回）还给调用者。

函数调用的机器级表示

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

ISA寄存器使用约定

n 调用者保存寄存器

p 调用者负责按需保存，被调用者可直接使用

n X86: IA32 EAX、EDX、ECX MIPS: \$t0~\$t9, \$a0~\$a3

n 被调用者保存寄存器

p 被调用者负责按需保存、返回之前恢复它们的值

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

n 保存方法（堆栈）

p pusha popa （将所有寄存器压栈/出栈）

p 也可按需保存恢复特定寄存器

为减少开销，优先使用哪些寄存器

中断处理程序没有调用者，如何处理

|| Intel 函数参数传递

n X86参数传递

- p 栈帧

n Linux IA-64参数传递

- p 先rdi, rsi, rdx, rcx, r8 和 r9 , 浮点数 xmm0-xmm7

- p 剩余的由右向左依次入栈

n Windows IA-64参数传递

- p rcx, rdx, r8, r9 , 浮点数 xmm0-xmm3

- p 剩余由右向左依次入栈

ABI (application binary interface)

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



计算机组成原理



五、指令系统



|| 本章主要内容

■ 5.1 指令系统概述

n 5.2 指令格式

n 5.3 寻址方式

n 5.4 指令类型

n 5.5 指令格式设计

n 5.6 CISC与RISC

n 5.7 指令系统举例



指令系统基本概念

n 机器指令（指令）

- p 计算机能直接识别、执行的某种操作命令

n 指令系统（指令集）

- p 一台计算机中所有机器指令的集合

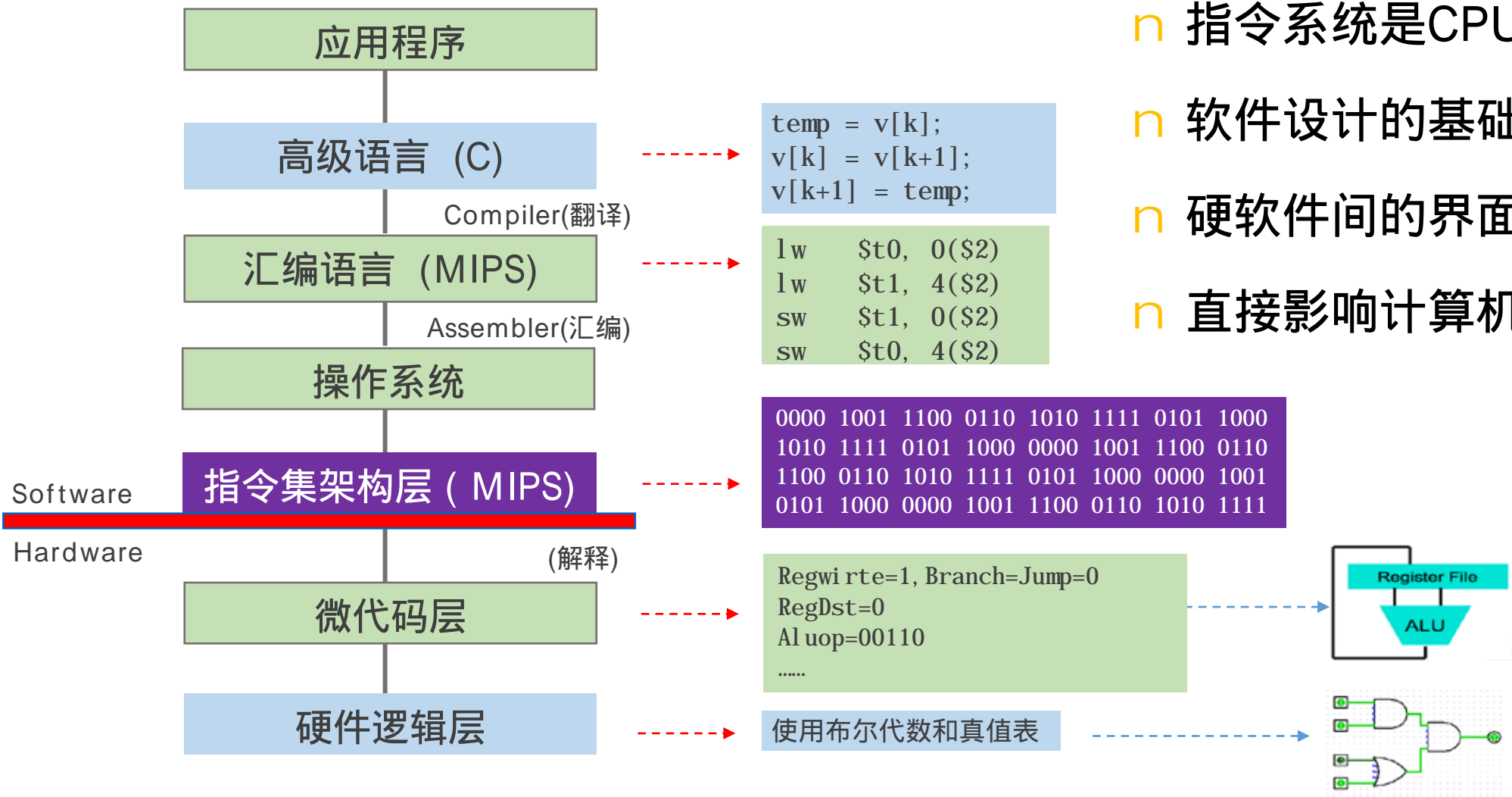
n 系列机

- p 基本指令系统相同，基本系统结构相同的计算机

- u IBM , PDP-11 , VAX-11 , Intel-x86

- p 解决软件兼容的问题

计算机指令系统层次



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

|| 本章主要内容

n 5.1 指令系统概述

n **5.2 指令格式**

n 5.3 寻址方式

n 5.4 指令类型

n 5.5 指令格式设计

n 5.6 CISC与RISC

n 5.7 指令系统举例



指令格式

n 表示一条指令的机器字，称为指令字，简称指令

n 指令格式：用二进制代码表示指令的结构形式

p 指令要求计算机处理什么数据？

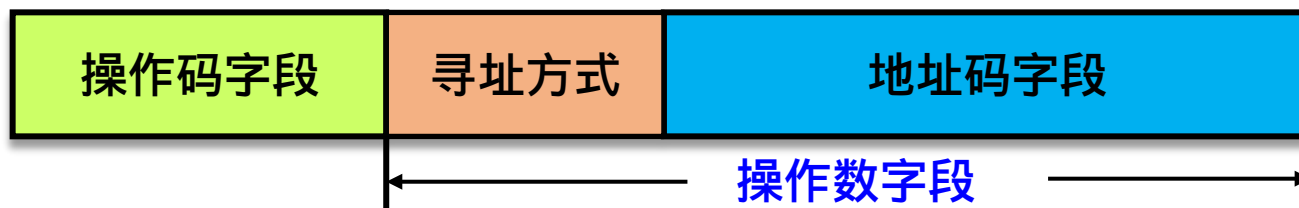
操作数字段需要解决的问题

p 指令要求计算机对数据做什么处理？

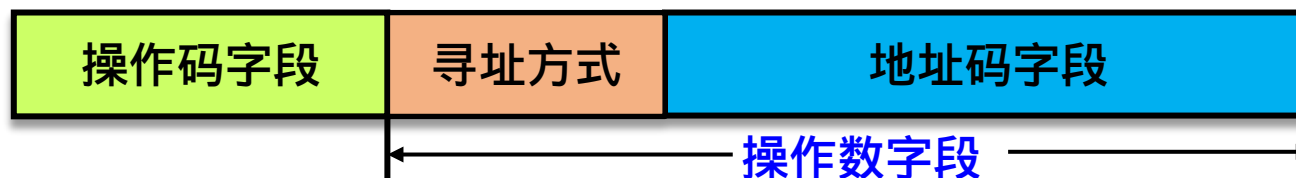
p 计算机怎样才能得到要处理的数据？

操作码字段需要解决的问题

寻址方式需要解决的问题



操作码(OP)与地址码(AC)



n 操作码字段长度决定指令系统规模

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

n 操作数字段可能有多

- p 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
- p 地址码字段 作用及影响、长度和寻址方式有关

指令字长度

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



指令地址码

三地址指令



(A2) OP (A3) A1

二地址指令



(A1) OP (A2) A1

单地址指令



(AC) OP (A1) AC

零地址指令



如停机、空操作、开关中断等

扩展操作码

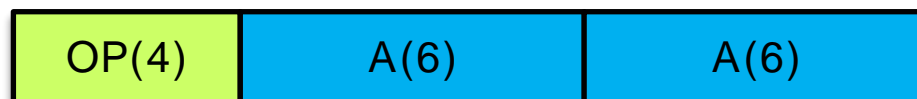


- n 3种指令操作码公共部分不得重叠，否则无法区分，译码
- n 设双操作数指令数为 k ，显然 $k < 2^8$
- n $2^8 - k$ 为多余状态，可用于表示其他类型指令
- n 可用于单操作数指令的条数 = $(2^8 - k) * 2^{12}$ ， 2^{12} 是多余12位组合

扩展指令举例

- n 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？

双操作数15



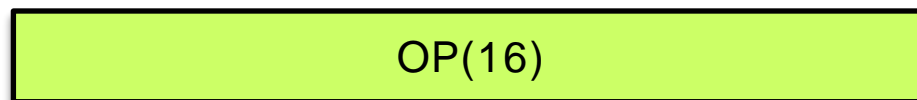
$$2^4$$

单操作数34



$$(2^4 - 15) * 2^6$$

无操作数?



$$((2^4 - 15) * 2^6 - 34) * 2^6$$

|| 本章主要内容

n 5.1 指令系统概述

n 5.2 指令格式

■ 5.3 寻址方式

n 5.4 指令类型

n 5.5 指令格式设计

n 5.6 CISC与RISC

n 5.7 指令系统举例



|| 寻址方式

n 寻找指令或操作数有效地址的方式

p 指令寻址

- u 顺序寻址、跳跃寻址

p 操作数寻址

- u 立即寻址、直接寻址

- u 间接寻址、寄存器寻址

- u 寄存器间接寻址、相对寻址

- u 基址\变址寻址、复合寻址

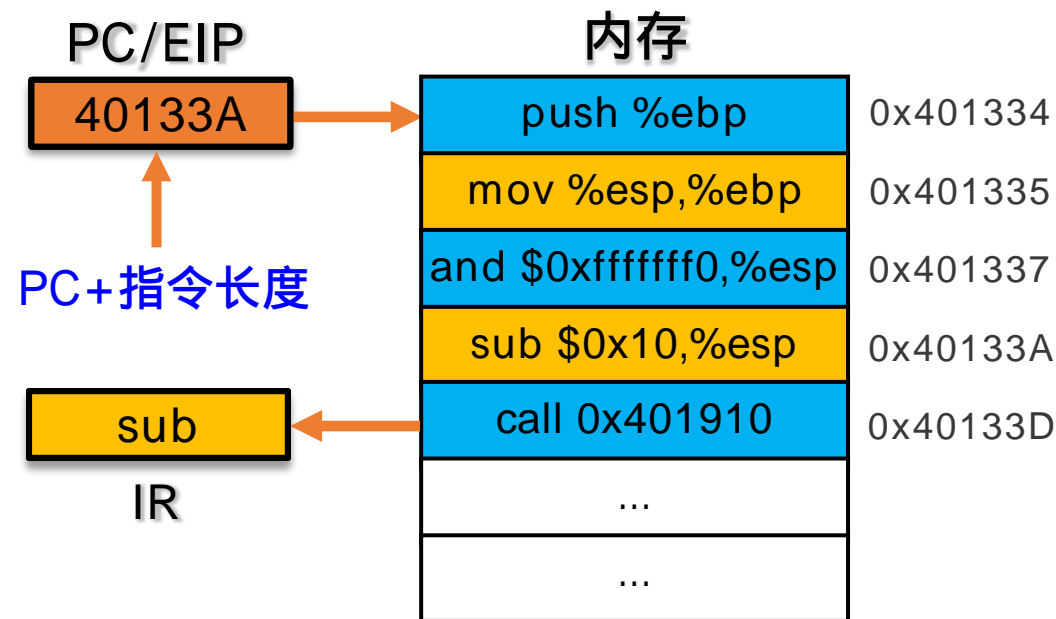
顺序寻址

顺序寻址

- 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， $PC = PC + \text{当前指令字节长度}$



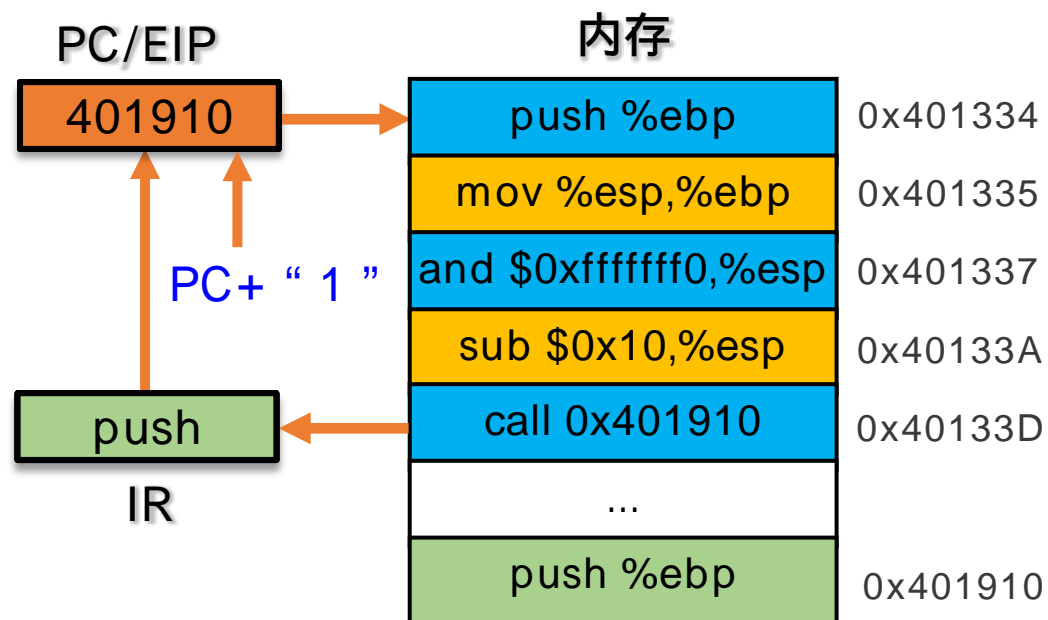
$M[pc++] \rightarrow IR$

|| 跳跃寻址

n **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

p 下条指令地址不是PC++得到，而是由指令本身给出

p 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) à PC

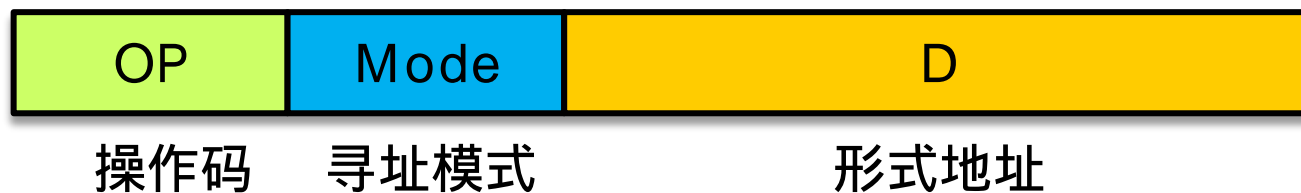
操作数的寻址方式

n 形成操作数有效地址的方法

p 单地址指令地址码的构成: mode , D

p 实际有效地址为 E, 实际操作数 S

p $S = (E)$



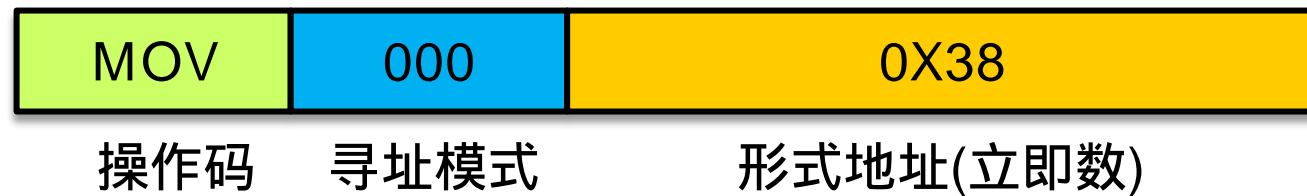


立即寻址

n 地址码字段是操作数本身

p S=D

p 例: MOV AX,38H (38H AX)

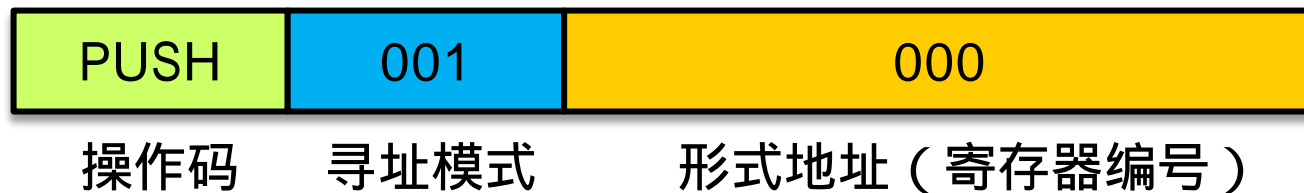


|| 寄存器寻址(Register Addressing)

n 操作数在CPU的内部寄存器中.

p AX,BX,CX,DX

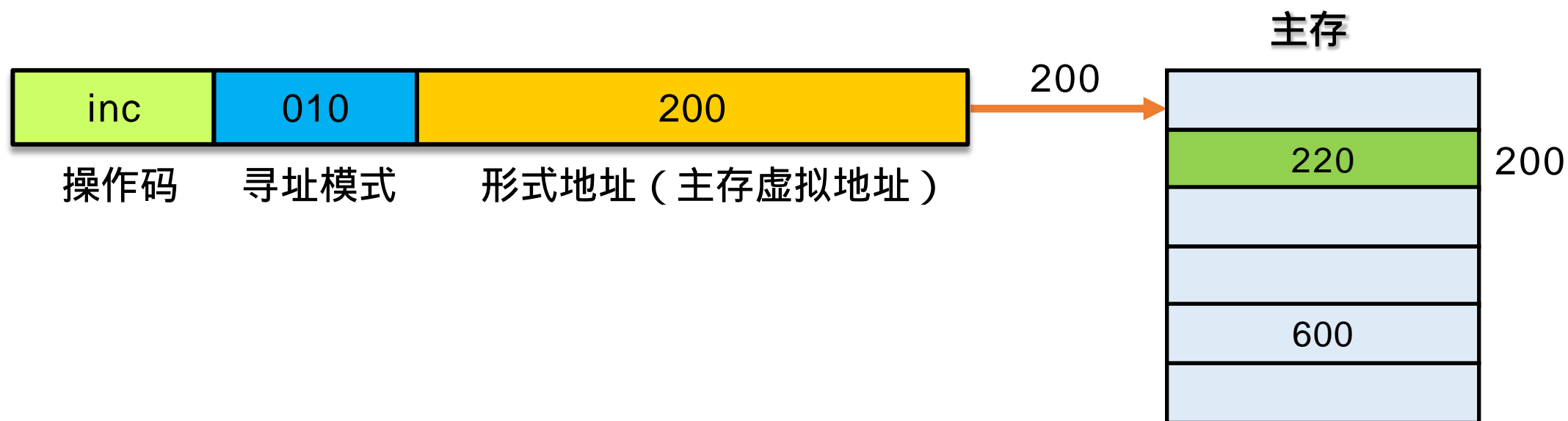
p PUSH AX E=R



直接寻址(Direct Addressing)

n 地址码字段直接给出操作数在内存的地址. $E=D$

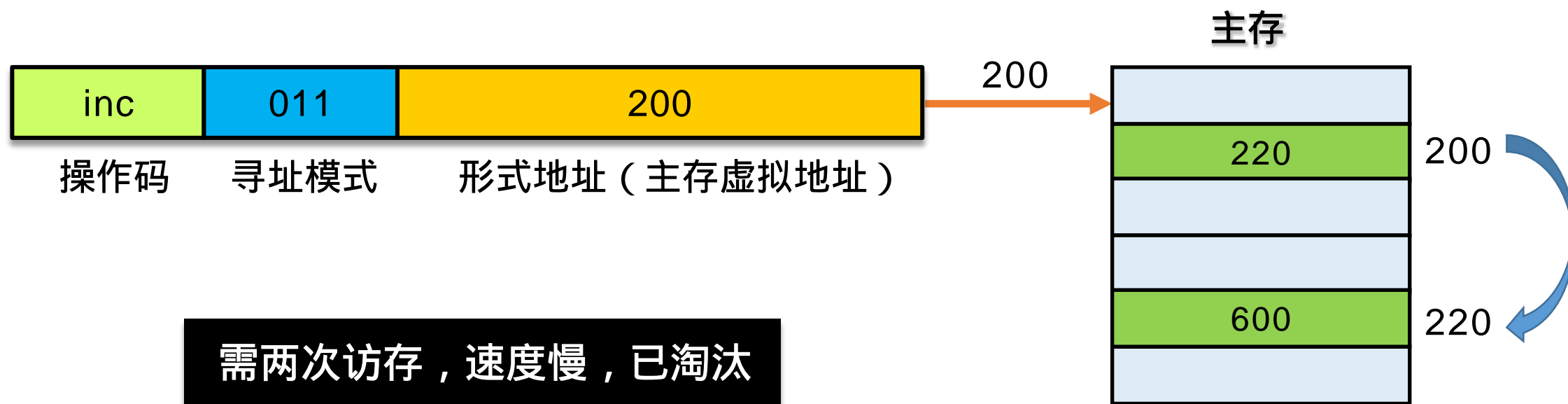
n inc [200]



间接寻址(Indirect Addressing)

n D单元的内容是操作数地址, D是操作数地址的地址

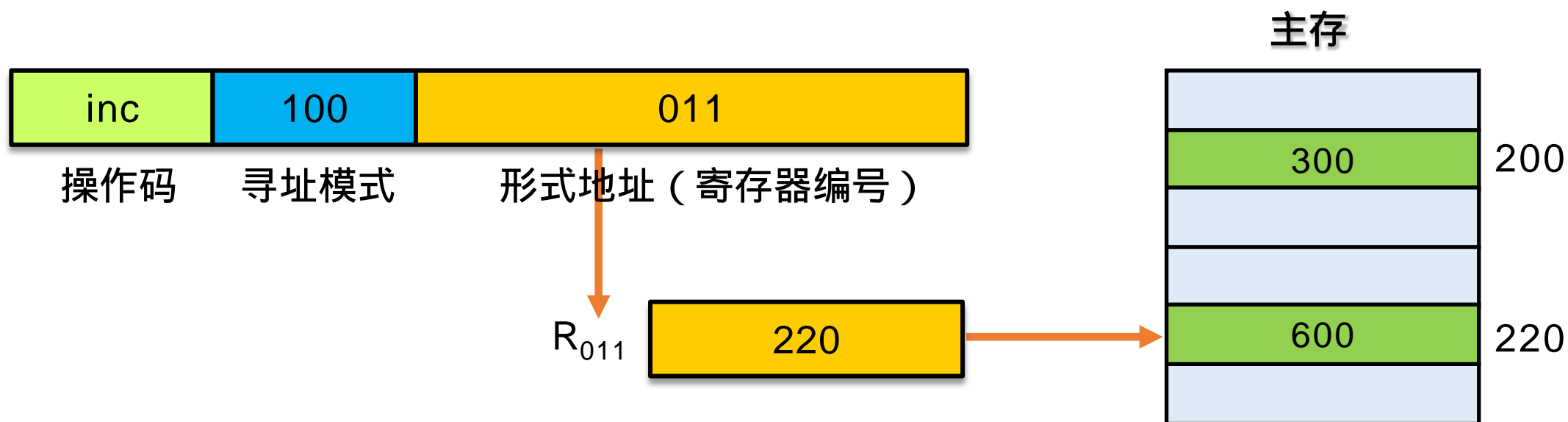
n $E=(D)$ $S=((D))$



寄存器间接寻址 (Register Indirect Addressing)

n D单元的内容是操作数的地址,D是操作数地址的地址

n $E=(R)$ inc [BX]

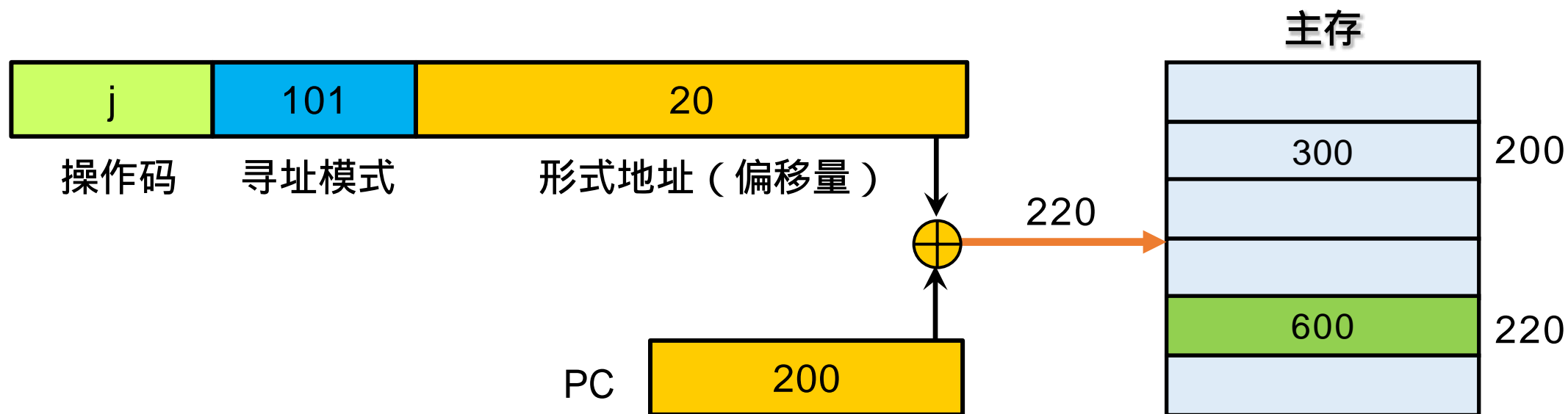


相对寻址 (Relative Addressing)

n 指令中的D加上PC的内容作为操作数的地址.

n $E = D + (PC)$

(PC)+D 还是(PC)+1+D?

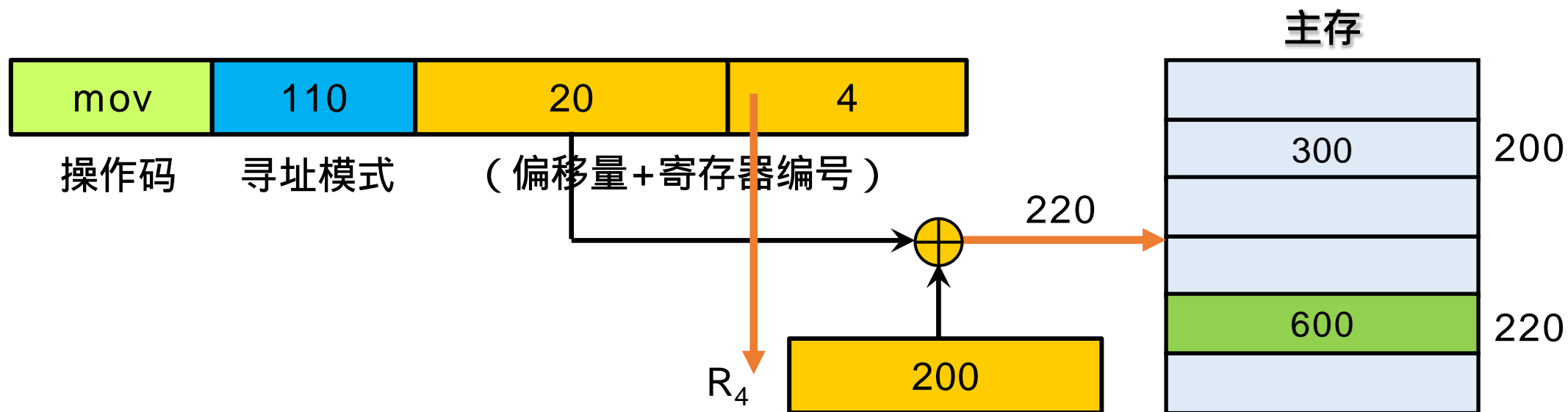


基址/变址寻址

n 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

n $E = D + (R)$

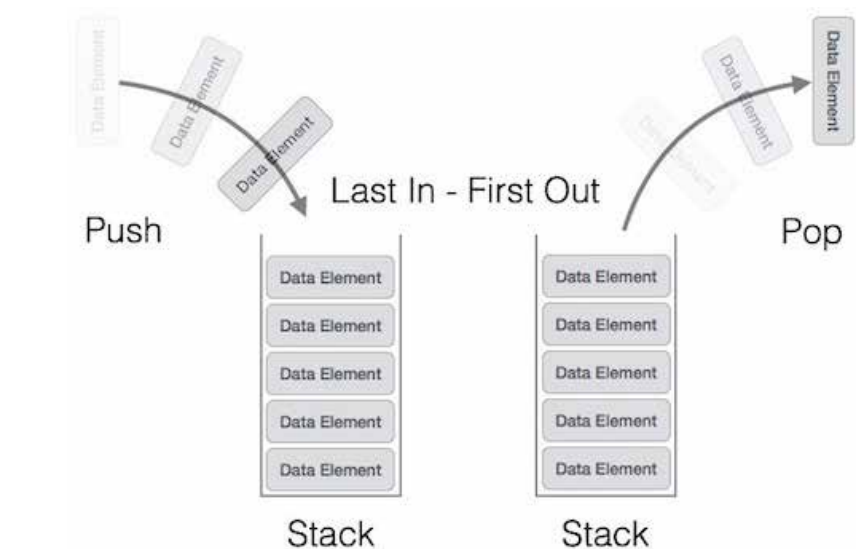
n `MOV AX, 32[SI]` SI, DI 都称为变址寄存器



堆栈寻址方式

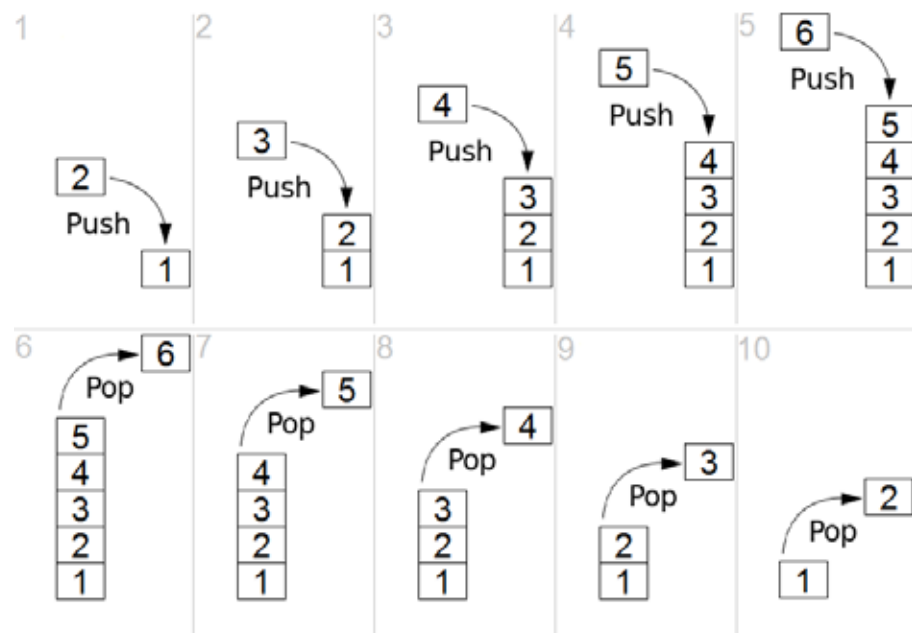
n 硬件堆栈（寄存器串联堆栈）

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

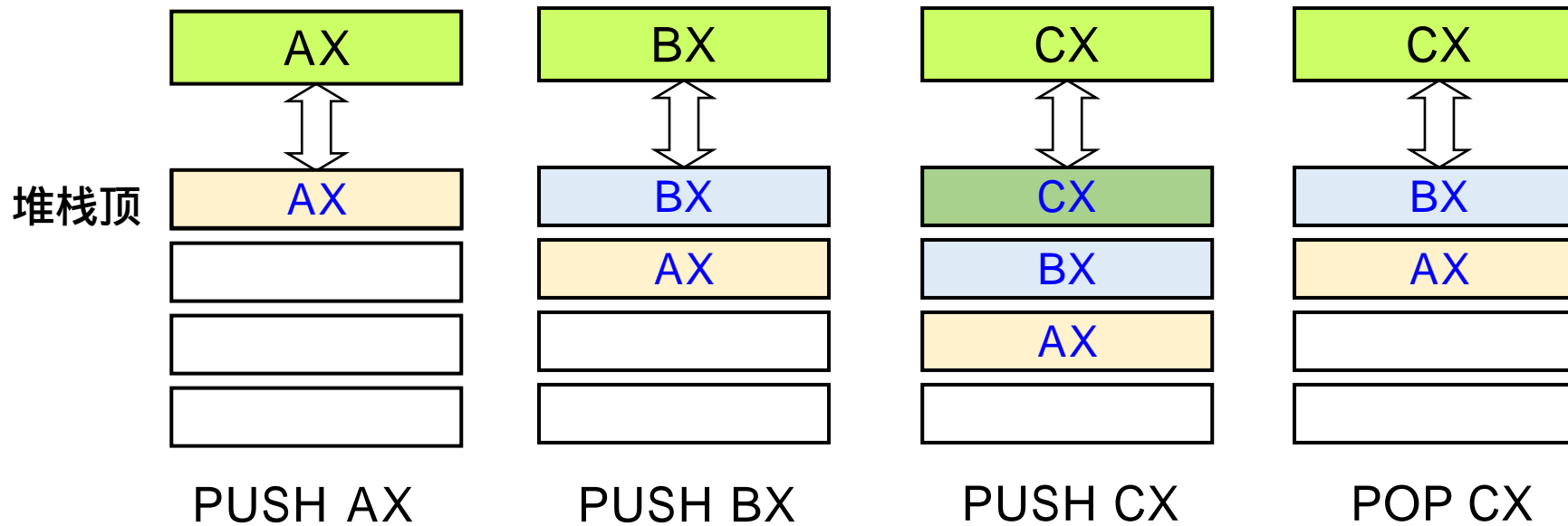


n 软件堆栈（内存堆栈）

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



硬件堆栈

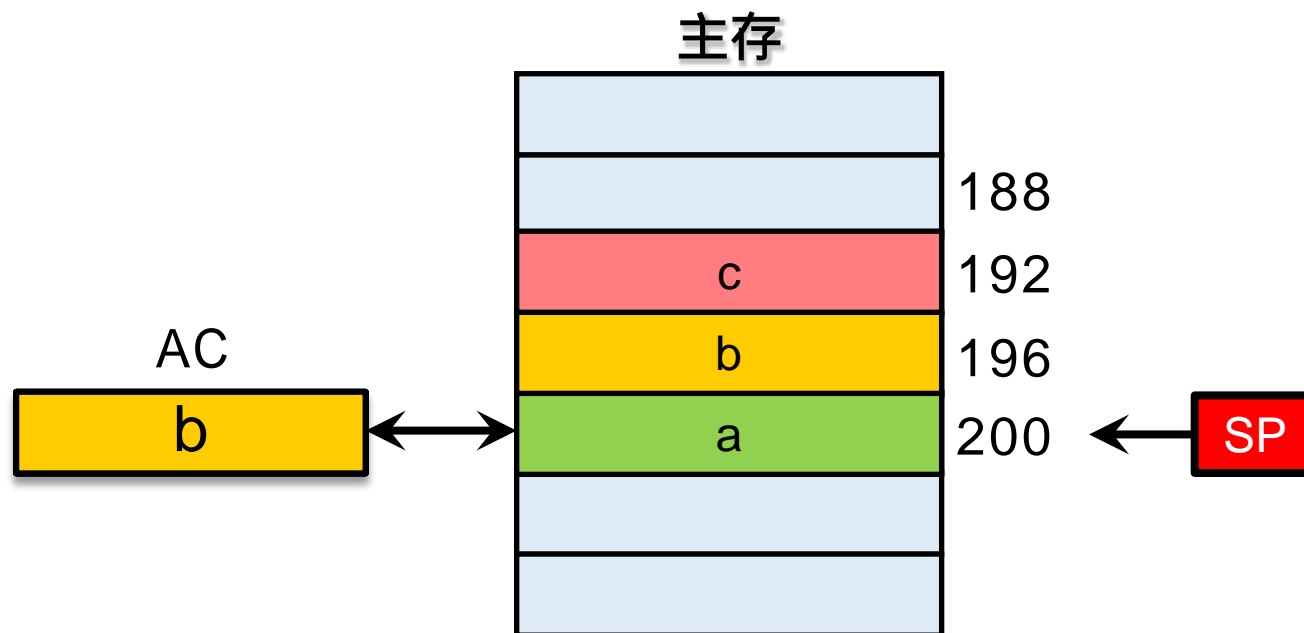


栈顶不动，数据移动

内存堆栈

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

n 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		实地址E	寻址范围
	操作码	寻址模式	形式地址D			
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

指令分类方法

n 按计算机系统的层次结构分类

p 微指令、机器指令、宏指令

n 按操作数物理位置分类

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

n 按指令长度分类

p 定长指令，变长指令

n 按操作数个数分类

p 四地址、三地址、二地址、单地址、零地址

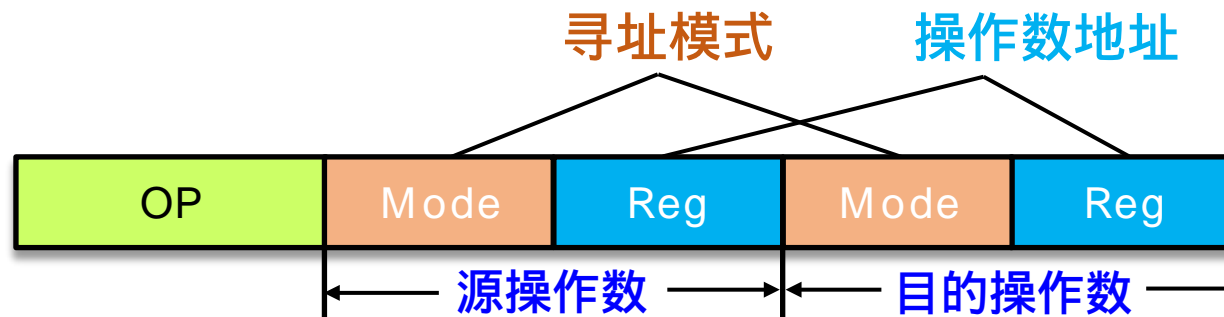
n 按指令功能分类

|| 本章主要内容

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



指令格式设计



- n 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
- n 根据对操作数的要求确定地址码字段的个数
- n 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
- n 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 OP字段需7位 ($2^7=128$)
- 4种寻址方式 寻址方式位需2位
- 单字长单地址 地址码长度 = $16 - 7 - 2 = 7$ 位



指令格式设计举例

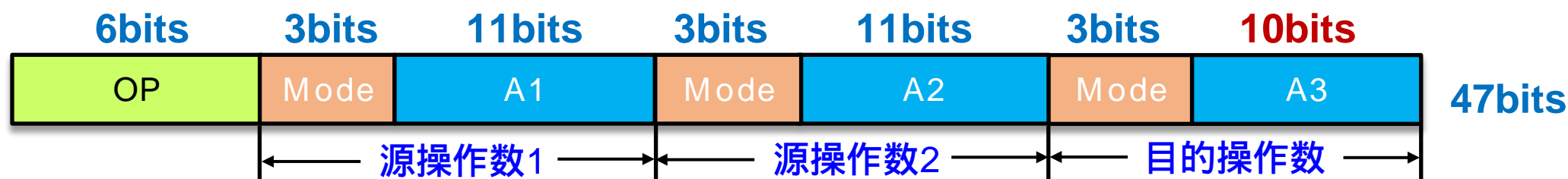
例2. 分析以下指令格式及寻址方式特点？



- n 1) 二地址指令；
- n 2) 操作码可指定16条指令；
- n 3) 源和目的均有8种寻址方式；
- n 4) 源地址寄存器和目的地址寄存器均有8个；
- n 5) 可寻址范围为1~64K (与机器字长有关)

指令格式设计举例

- 例3. 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

2010研究生统考例题

例4. 某计算机字长为16位，主存地址空间大小为128KB，按字编址。采用单字长指令格式，指令各字段定义如图，转移地址采用相对寻址方式，相对偏移量用补码表示。寻址方式如图。



M _s /M _d	寻址方式	助记符	含义
000B	寄存器直接	R _n	操作数=(R _n)
001B	寄存器间接	(R _n)	操作数=((R _n))
010B	寄存器间接，自增	(R _n)+	操作数=((R _n)), (R _n)+1→(R _n)
011B	相对	D(R _n)	转移目标地址=(PC)+(R _n)

注(x)表示存储器地址x或寄存器x的内容

(1) 该指令系统最多可有多少条指令？该计算机最多有多少个通用寄存器？存储器地址寄存器MAR和存储器数据寄存器MDR至少需要多少位？

2010研究生统考例题

某计算机字长为16位，主存地址空间大小为128KB，按字编址。采用单字长指令格式，指令各字段定义如图，转移地址采用相对寻址方式，相对偏移量用补码表示。寻址方式如图。



M _s /M _d	寻址方式	助记符	含义
000B	寄存器直接	R _n	操作数=(R _n)
001B	寄存器间接	(R _n)	操作数=((R _n))
010B	寄存器间接，自增	(R _n)+	操作数=((R _n)), (R _n)+1→(R _n)
011B	相对	D(R _n)	转移目标地址=(PC)+(R _n)

注(x)表示存储器地址x或寄存器x的内容

(2) 转移指令的目标地址范围是多少？

2010研究生统考例题

某计算机字长为16位，主存地址空间大小为128KB，按字编址。采用单字长指令格式，指令各字段定义如图，转移地址采用相对寻址方式，相对偏移量用补码表示。寻址方式如图。

M_s/M_d	寻址方式	助记符	含义
000B	寄存器直接	R_n	操作数= (R_n)
001B	寄存器间接	(R_n)	操作数= $((R_n))$
010B	寄存器间接，自增	$(R_n)+$	操作数= $((R_n)), (R_n)+1 \rightarrow (R_n)$
011B	相对	$D(R_n)$	转移目标地址= $(PC)+(R_n)$

(3) 若操作码0010B表示加法操作，助记符为add，寄存器R4，R5的编号分别为100B和101B，R4的内容为1234H，R5的内容为5678H，地址1234H中的内容为5678H，地址5678H中的内容为1234H，则汇编语句add (R4),(R5)+ 逗号前为源操作数，逗号后为目的操作数，对应的机器码是多少？用十六进制表示。该指令执行以后，哪些寄存器和存储单元的内容会发生改变？改变后的内容是什么？

|| 本章主要内容

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



指令系统发展方向

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



精减指令系统(RISC)

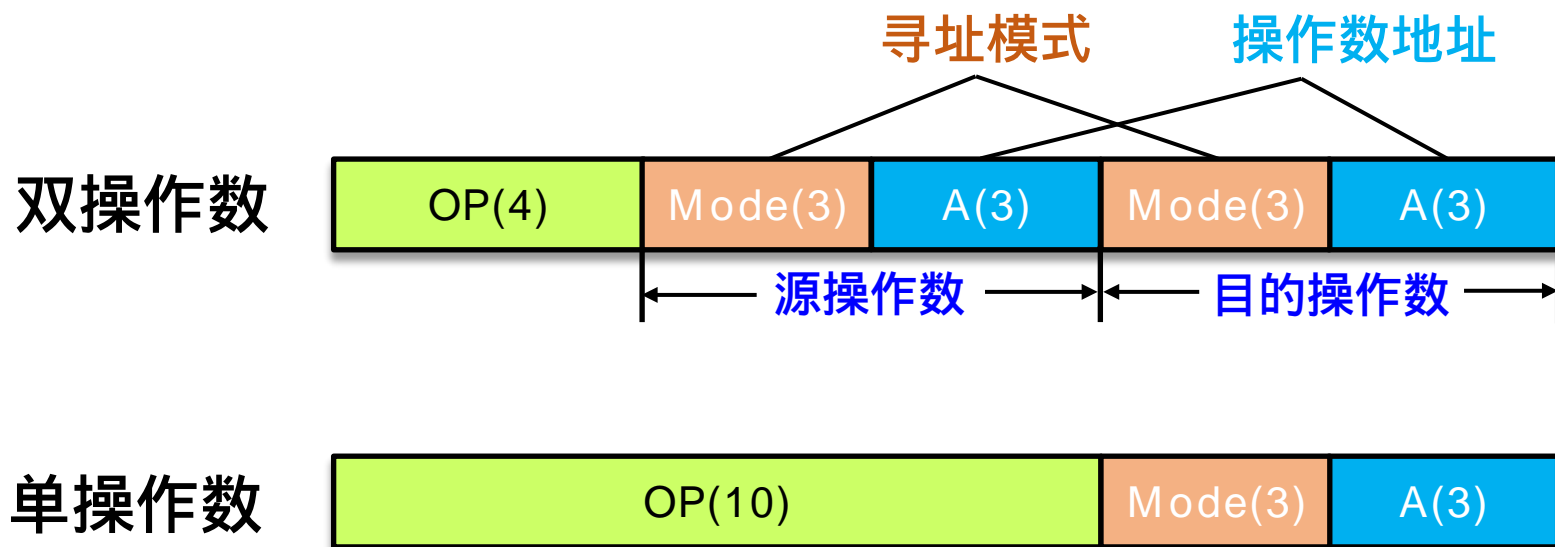
- n 指令条数少，只保留使用频率最高的简单指令，指令定长
 - p 便于硬件实现，用软件实现复杂指令功能
- n Load/Store架构
 - p 只有存/取数指令才能访问存储器，其余指令的操作都在寄存器之间进行
 - p 便于硬件实现
- n 指令长度固定，指令格式简单、寻址方式简单
 - p 便于硬件实现
- n 寄存器数量多（32~192）
 - p 便于编译器实现
- n 一个机器周期完成一条机器指令
- n RISC CPU采用硬布线控制，CISC采用微程序

|| 本章主要内容

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



PDP-11指令格式



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



|| PDP-11指令集特点



n 机器字长16位

n 单字长，双字长，三字长指令

p 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

n 8种寻址方式

n 8个16位寄存器r0~r7，有条件状态寄存器PSW

p r0~r5通用寄存器，r6为栈指针SP，r7为程序计数器PC

n 较好的规整性，典型的扩展操作码指令



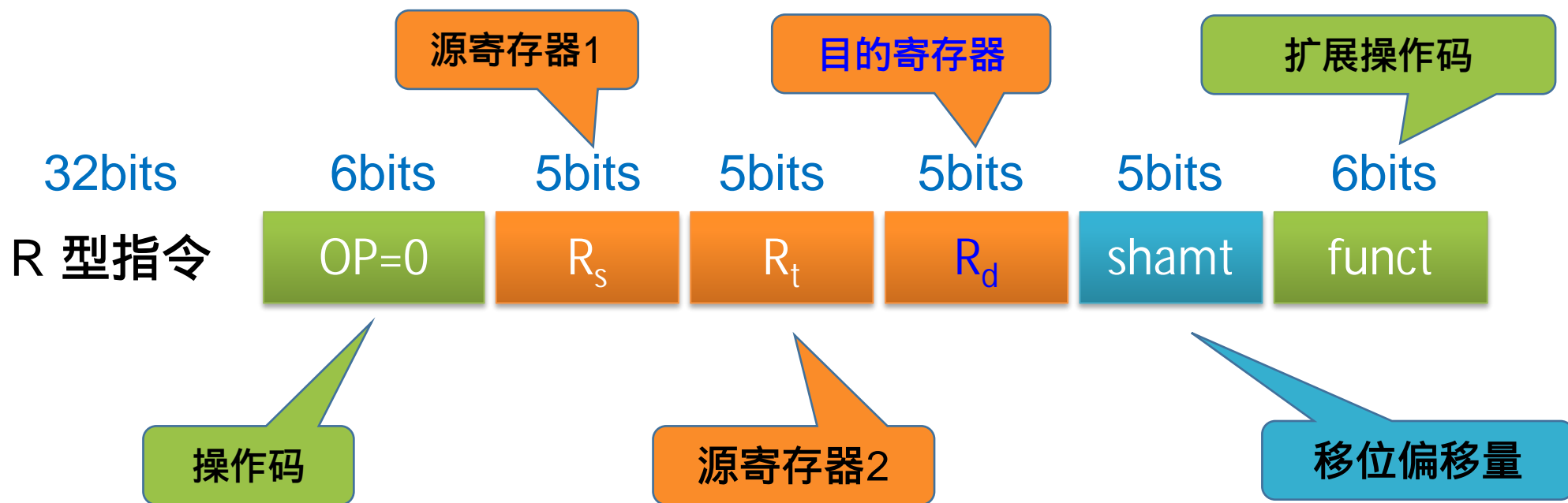
PDP-11寻址方式

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	$index(R_i)$	操作数地址=寄存器的值+16位index
7	变址 间址	$@index(R_i)$	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)

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 (R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
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

机器码 0x2538820

MIPS指令格式



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寻址方式总结

n 寄存器寻址

n 变址寻址

n 立即数寻址

n PC相对寻址 `beq reg1,reg2,offset`

p $PC + 4 + 16\text{位偏移地址左移两位}$

p 字地址变字节地址

n 伪直接寻址 `J label`

PC 高4位

26位立即数

00

开源MIPS仿真器，汇编器

The screenshot displays the MARS MIPS simulator interface. The main window shows assembly code for a program that calculates the sum of integers from 1 to n. The code is as follows:

```

1  # This program computes and displays the sum of integers from 1 up to n,
2  # where n is entered by the user.
3  #
4
5  .globl    main
6
7  .data
8
9  # program output text constants
10 prompt:
11 .asciz    "Please enter a positive integer: "
12 result1:
13 .asciz    "The sum of the first "
14 result2:
15 .asciz    " integers is "
16 newline:
17 .asciz    "\n"
18
19 .text
20
21 # main program
22 #
23 # program variables
24 # n: $s0
25 # sum: $s1
  
```

The interface includes a menu bar (File, Edit, Run, Settings, Tools, Help) and a toolbar with various icons. The right panel shows the registers, and the bottom panel shows the Mars Messages / Run I/O window.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffefffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000



华中科技大学
计算机科学与技术学院
School of Computer Science & Technology, HUST

THANKS

计算机组成原理

