Lecture 5 指令系统体系结构3

1. MIPS寻址

寻址：指令如何识别指令中操作数

MIPS寻址种类

- 立即数寻址 **Immediate addressing**

  addi $s0, $s1, 5
- 寄存器寻址 **Register addressing**

  add $s0, $s1, $s2
- 基址寻址 **Base addressing**

  lw $s0, 0($s1)
- **PC**相对寻址 **PC-relative addressing**

  bne $s0, $s1, EXIT
- 伪-直接寻址 **Pseudo-direct addressing**

  j EXIT

立即数寻址

指令类型

- addi，subi，andi，ori

大多数常数都很小

- 16位立即数就足够了

大常数立即数寻址

如果指令数非常大，大于16bit了，我们在导入数据的时候会使用**lui指令**

lui指令：读取立即数高位指令 load upper immediate，允许后续指令设置常数的低16位

例如，将下列数据导入到寄存器中

```
0000 0000 0111 1101 0000 0000 0000 0000
```

我们先使用

```
lui $s0, 61
001111 00000 01000 0000 0000 1111 1111
```

将高比特16位导入$s0的高比特位，低16位用0填充

然后使用

```
ori $s0, $s0, 2304
```

将低比特位导入

寄存器寻址

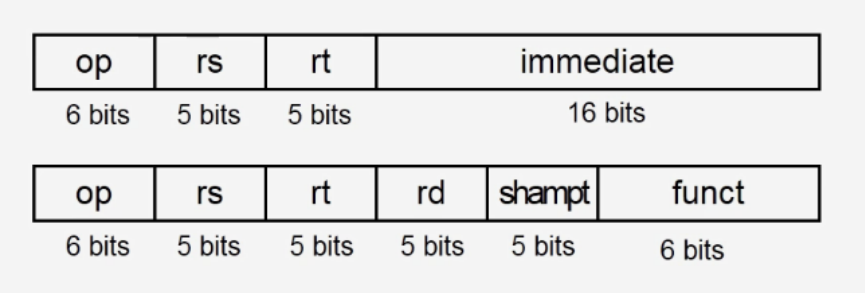
如果输入包含了寄存器，那么就是寄存器寻址

大部分指令（除了j，jr）其它基本都用寄存器寻址

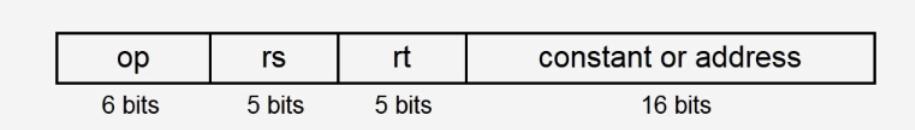
指令类型

- add，addi，sub，subi，lw...

两种指令格式



分支寻址（pc相对寻址）



指令类型

- bne，beq...

事实上，分支寻址就是基于PC相对寻址实现的，PC记录了这次指令执行完后的下一个指令的位置，MIPS寻址实际上是**相对于下一条指令的地址（PC+4）**而不是相对于当前指令（PC）

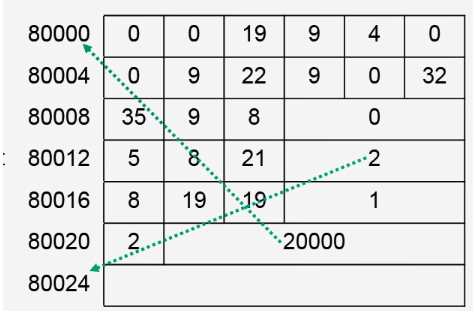
constant or address部分记录了PC将会跳转到哪里

由于一个指令需要占4个字节

那么目标地址的位置为

$$Target\ address = PC + address \times 4$$

例如



从80012行开始，此时PC = 80016， address = 2

所以下一条指令的地址为 80016 + 2 × 4 = 80024

远距离条件分支转移

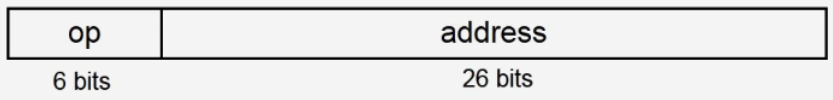
假设寄存器$s0与寄存器$s1的值相等时需要跳转，可以使用如下指令

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

可以用两条指令替换上面的指令，获得更远的转移距离

```
    bne $s0, $s1, L2
    j   L1
L2:
    ...
```

伪-直接寻址



指令类型

- j，jal

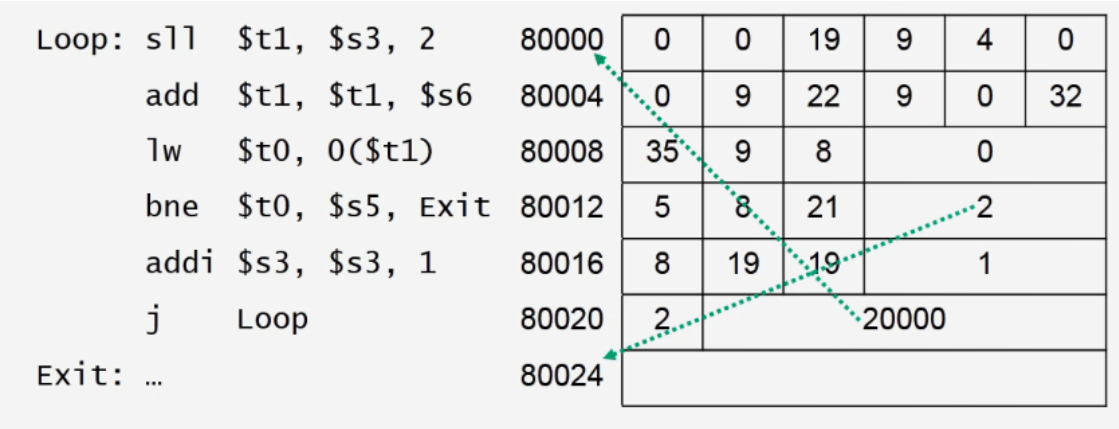
MIPS跳转指令虽然是**26位**，但是由于其单位是**字而不是字节**，所以我们在计算的时候会**乘4**来得到真正的地址，也就是会右移动2位，达到补充到28位的效果

由于PC是32位，所以有4位必须来自跳转指令以外的其它地方，MIPS跳转指令仅仅代替PC的低28位，**而高4位保持不变**

首先我们要取PC计数器的高4个比特，也就是一个字节作为高位

保证PC的前4位不变，上下跳转

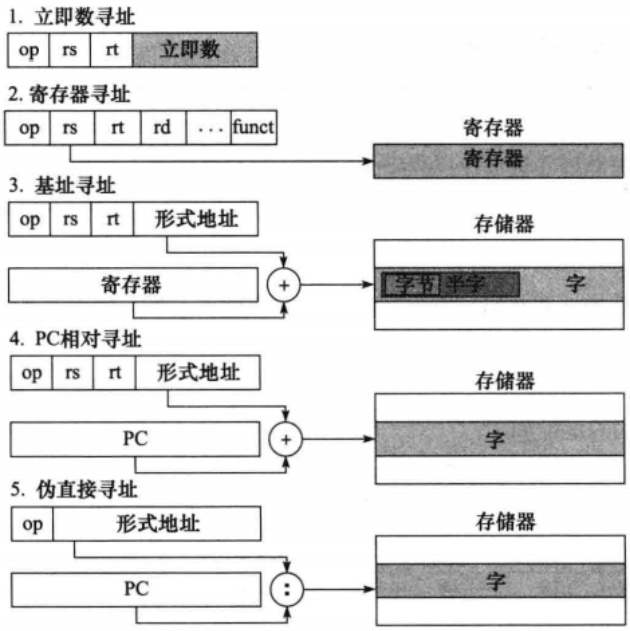
$$Target\ address = PC[31,30,29,28] + (address \times 4)(\text{右移两个}0)$$



80020行，程序想跳转到80000行，则address = 80000/4 = 20000

如果指令太远了，beq和bne都表示不了的时候，**程序会自动使用j命令来进行更远距离的跳转**，我们不需要手动调用

寻址模式总结

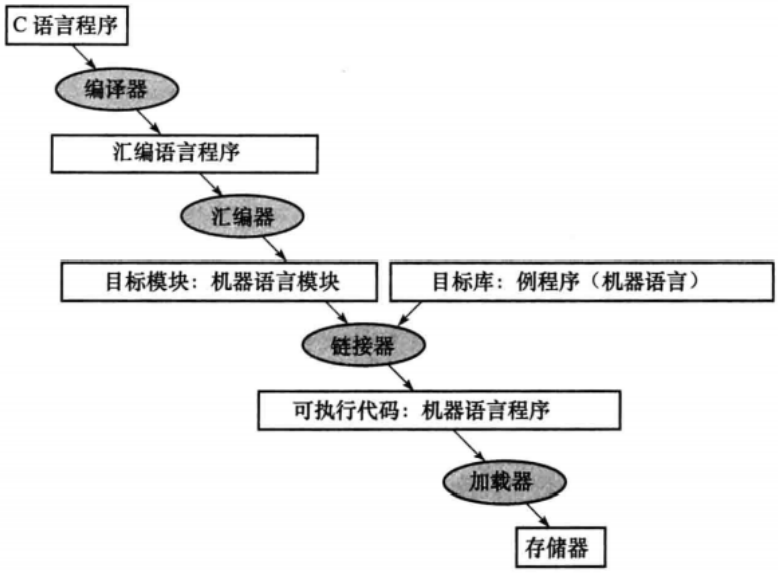


MIPS 5 种寻址模式的说明。阴影部分为操作数。模式 3 的操作数在内存中，而模式 2 的操作数是寄存器。注意，读数和存数对字节、半字或字有多种版本。模式 1 的操作数是指令自身的 16 位字段。模式 4 和模式 5 寻址的指令在内存中，模式 4 把 16 位地址左移 2 位与 PC 相加，而模式 5 把 26 位地址左移 2 位与 PC 计数器的高 4 位相连。注意，一种操作可能可以使用多种寻址模式，例如，加法可以使用立即数寻址（addi）和寄存器寻址（add）

- **立即数寻址(immediate addressing)**
  操作数是位于指令自身中的常数
- **寄存器寻址(register addressing)**
  操作数是寄存器
- **基址寻址(base addressing)或偏移寻址(displacement addressing)**
  操作数在内存中,其地址是指令中基址寄存器和常数的和

- **PC相对寻址(PC-relative addressing)**
  地址是PC和指令中常数的和
- **伪直接寻址(pseudodirect addressing)**
  跳转地址由指令中26位字段和PC高位相连而成

2. 翻译并执行程序



C 语言的翻译层次。用高级语言编写的程序首先需要被编译成为汇编语言，然后被汇编成机器语言组成的目标模块。链接器将多个模块和库程序组合在一起解析所有的引用。加载器将可执行程序加载到内存的适当位置，然后处理器就可以执行了。为了加快翻译的速度，某些步骤被跳过或和其他步骤组合在一起。一些编译器直接产生目标模块，一些系统使用带链接功能的加载器直接完成后面两步。为了确定文件的类型，UNIX 使用文件的后缀，x.c 代表 C 源文件，x.s 表示汇编文件，x.o 表示目标文件，x.a 表示静态链接库，x.so 表示动态链接库，默认情况下，a.out 表示可执行文件。MS-DOS 使用后缀 .C，.ASM，.OBJ，.LIB，.DLL 和 .EXE 来完成同样的功能

编译器

编译器将C程序转换成一种机器能理解的符号形式的**汇编程序语言**

汇编语言：一种符号语言,能被翻译成二进制的机器语言

汇编器

汇编语言对于高层次软件是一个接口，它也存在一些语言作为机器语言常见代变种，在汇编语言中的存在简化了程序转换和变成，这类指令称为**伪指令pseudoinstruction**

例如

```
1  move $t0, $t1
```

实时上，MIPS体系结构中不存在这条指令，但是汇编器可以将这条汇编语言指令转化成等价的如下机器语言指令

```
1  add $t0, $zero, $t1
```

之前提到的**远距离条件分支转移**，在某些情况下也是一种伪指令

伪指令使MIPS拥有比硬件所实现的更为丰富的汇编语言指令集，唯一的代价是**保留了一个由汇编器使用的寄存器$at**

汇编器的主要任务是**汇编成机器代码**，汇编器将汇编语言程序转换成**目标文件object file**，它包括**机器语言指令，数据和指令正确放入内存所需要的信息**

目标文件 Object File

UNIX系统中的目标文件通常包含以下6个不同的部分

- **目标文件头**：描述目标文件其他部分的大小和位置
- **代码段**：包含机器语言代码
- **静态数据段**：包含在程序生命周期内分配的数据。(UNIX系统允许程序使用静态数据，它存在于整个程序中，也允许使用动态数据，它随程序的需要而增长或缩小
- **重定位信息**：标记了一些在程序加载进内存时依赖于绝对地址的指令和数据
- **符号表**：包含未定义的剩余标记，如外部引用
- **调试信息**：包含一份说明目标模块如何编译的简明描述，这样，调试器能够将机器指令关联到C源文件，并使数据结构也变得可读

链接器

对于标准库程序，汇编全部代码非常浪费，所以链接器的出现，就是**把所有独立汇编的机器语言程序拼接在一起**

- 将代码和数据模块象征性地放入内存
- 确定数据地址和指令标签
- 修补内部和外部引用

链接器产生一个**可执行文件**，它可以在一台计算机上运行，通常这个文件与目标文件有相同的格式，但是它不包含未解决的引用（库程序可以包含）

示例

将下面的两个目标文件链接，给出最终可执行文件中前几条指令对应的更新过的地址

为了便于理解，我们使用汇编语言来表示指令，在实际文件中，这些指令由数字表示

注意目标文件中，我们已将必须在链接进程中更新的地址和标记高亮显示了，分别是**引用过程A和过程B的地址的指令**，以及**引用数据字x和Y的地址的指令**

目标文件头			
	名字	过程 A	
	正文大小	100_{16}	
	数据大小	20_{16}	
代码段	地址	指令	
	0	lw $a0,0($gp)	
	4	jal 0	
	…	…	
数据段	0	(X)	
	…	…	
重定位信息地址	地址	指令类型	依赖
	0	lw	X
	4	jal	B
符号表	标记	地址	
	X	—	
	B	—	

目标文件头			
	名字	过程 B	
	正文大小	200_{16}	
	数据大小	30_{16}	
代码段	地址	指令	
	0	sw $a1,0($gp)	
	4	jal 0	
	…	…	
数据段	0	(Y)	
	…	…	
重定位信息地址	地址	指令类型	依赖
	0	sw	Y
	4	jal	A
符号表	标记	地址	
	Y	—	
	A	—	

由内存分配可知，代码段的起始地址是 $40\ 0000_{16}$，数据段的起始地址是 $1000\ 0000_{16}$

过程A的正文被放置在第一个地址，而它的数据被放置在第二个地址

过程A的目标文件头表明其代码段的大小为 100_{16} 字节，而数据段的大小为 20_{16} 字节，这样过程B的代码段的开始地址就是 $40\ 0100_{16}$，数据段的开始地址就是 $1000\ 0020_{16}$

链接器更新指令的地址字段，它使用指令类型字段得到待编地址的格式

- jal类型是伪直接寻址，对于地址 $40\ 0000_{16}$ 处的jal，其它地址字段是 $40\ 0100_{16}$，而地址 $40\ 0104_{16}$ 处的jal地址字段是 $40\ 0000_{16}$
- sw指令对应的地址更加复杂，它们和基址寄存器有关，全局指针 $gp 的初始值为 $1000\ 8000_{16}$，为了得到地址 $1000\ 0000_{16}$，我们设lw的地址字段为8000_{16}

可执行文件：

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

加载器

把目标程序装载到内存中以准备运行的系统程序

可执行文件在磁盘中，操作系统可以将其读入内存并启动执行它

加载器的工作步骤如下

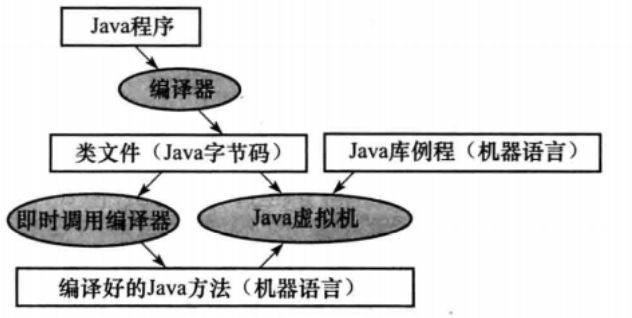
1. 读取可执行文件头来确定代码段和数据段的大小
2. 为正文和数据创建一个足够大的地址空间
3. 将可执行文件中的指令和数据复制到内存中
4. 把主程序的参数(如果存在)复制到栈顶
5. 初始化机器寄存器,将栈指针指向第一个空位置
6. 跳转到启动例程,它将参数复制到参数寄存器并且调用程序的main函数。当main函数返回时，启动例程通过系统调用exit终止程序

汇编语言翻译成机器码

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Java程序



Java程序会首先被编译成易于解释的指令序列——**Java字节码指令集**

而不是编译成目标计算机可识别的汇编语言

Java虚拟机JVM能够执行Java字节码文件，**解释器是用来模拟指令集体系结构的程序，**不需要再单独进行汇编

3. 基本算法的底层实现

排序

C代码

```c
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
            swap (v,j);
        }
    }
}
```

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

MIPS代码-swap

寄存器分配：

- $a0：v[]
- $a1：k
- $t0-$t2：调度使用的寄存器

```
swap:
    sll $t1, $a1, 2 # MIPS是按字节在内存中寻址的，字由4字节组成，因此需要把 t1 = a1 × 4
    add $t1, $a0, $t1   # t1 = Address- v + (k * 4)
    lw $t0, 0($t1)      # t0 = v[k]
    lw $t2, 4($t1)      # t2 = v[k+1]
    sw $t2, 0($t1)      # v[k] = t2
    sw $t0, 4($t1)      # v[k+1] = t0
    jr $ra
```

MIPS代码-sort

寄存器分配

- $a0：v[]
- $a1：k
- $s0：i
- $s1：j

保存寄存器值

```
sort:   addi    $sp, $sp, -20       # make room on stack for 5 registers
        sw      $ra,16($sp)# save $ra on stack
        sw      $s3,12($sp)         # save $s3 on stack
        sw      $s2,8($sp)# save $s2 on stack
        sw      $s1,4($sp)# save $s1 on stack
        sw      $s0,0($sp)# save $s0 on stack
```

过程体

移动参数	`move $s2, $a0 # copy parameter $a0 into $s2 (save $a0)` `move $s3, $a1 # copy parameter $a1 into $s3 (save $a1)`

循环外部	`move $s0, $zero # i = 0`
	`forltst:slt $t0, $s0, $s3 # reg $t0 = 0 if $s0 < $s3 (i < n)`
	`beq $t0, $zero, exit1 # go to exit1 if $s0 < $s3 (i < n)`

循环内部	`addi $s1, $s0, -1 # j = i - 1`
	`for2tst:slti $t0, $s1, 0 # reg $t0 = 1 if $s1 < 0 (j < 0)`
	`bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)`
	`sll $t1, $s1, 2 # reg $t1 = j * 4`
	`add $t2, $s2, $t1 # reg $t2 = v + (j * 4)`
	`lw $t3, 0($t2)# reg $t3 = v[j]`
	`lw $t4, 4($t2)# reg $t4 = v[j + 1]`
	`slt $t0, $t4, $t3 # reg $t0 = 0 if $t4 < $t3`
	`beq $t0, $zero, exit2 # go to exit2 if $t4 < $t3`

传递参数和调用	`move $a0, $s2 # 1st parameter of swap is v (old $a0)`
	`move $a1, $s1 # 2nd parameter of swap is j`
	`jal swap # swap code shown in Figure 2.25`

循环内部	`addi $s1, $s1, -1 # j -= 1` `j for2tst # jump to test of inner loop`
循环外部	`exit2: addi $s0, $s0, 1 # i += 1` `j for1tst # jump to test of outer loop`

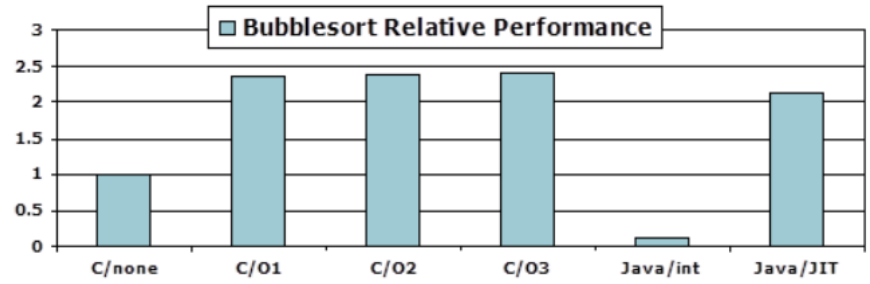
恢复寄存器的值

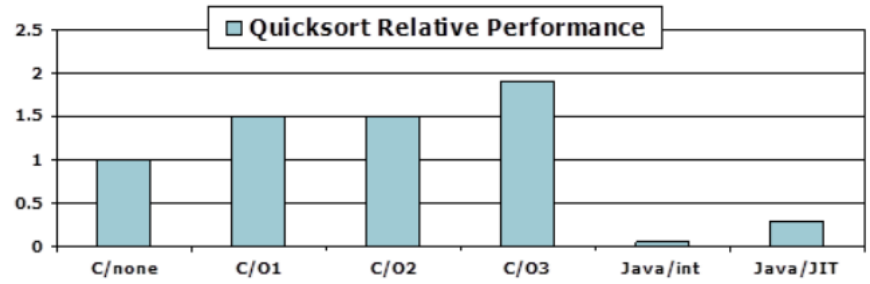
```
exit1:  lw      $s0,0($sp)          # restore $s0 from stack
        lw      $s1,4($sp)# restore $s1 from stack
        lw      $s2,8($sp)# restore $s2 from stack
        lw      $s3,12($sp)         # restore $s3 from stack
        lw      $ra,16($sp)         # restore $ra from stack
        addi    $sp, $sp,20         # restore stack pointer
```

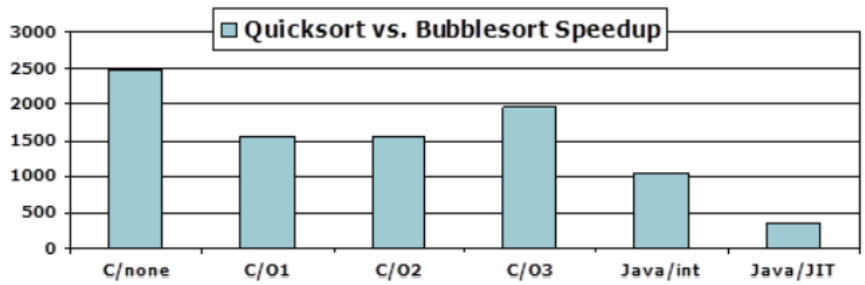
过程返回

```
        jr      $ra                 # return to calling routine
```

排序算法的效率







- 单独评价指令数和CPI不是很好的性能指标
- 编译器优化对算法很敏感
- Java/JIT编译的代码比JVM解释快
- 没有什么能修复愚蠢的算法

4. 其他的指令集

ARM

在嵌入式设备中最流行的指令集体系结构是ARM

ARM和MIPS相比

- MIPS有更多的寄存器
- ARM有更多寻址模式

ARM是用的最广的指令集

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

x86

经典的RISC指令集