第二章、指令：计算机的语言总结

1、数据对齐：

字的起始地址都是4的倍数

2、大端小端：

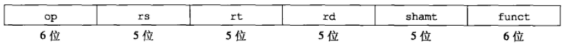
0xabcdef12大小端存放方式如下：

Little-Endian		Big-Endian	
Address	Data	Address	Data
12	ab	12	12
8	cd	8	ef
4	ef	4	cd
0	12	0	ab

3、指令格式：

3种，R型，I型，J型

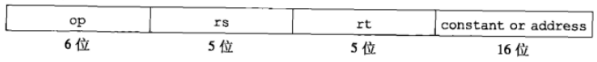
R：



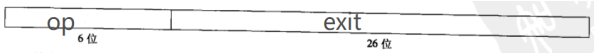
其中各字段名称及含义如下：

- op：指令的基本操作，通常称为**操作码**[○]。
- rs：第一个源操作数寄存器。
- rt：第二个源操作数寄存器。
- rd：用于存放操作结果的目的寄存器。
- shamt：位移量。（在 2.6 节中介绍移位指令和该术语，在此之前，指令都不使用这个字段，故此字段的内容为 0。）
- funct：功能。一般称为**功能码**（function code），用于指明 op 字段中操作的特定变式。

I：



J：



名称	字段						备注
字段大小	6 位	5 位	5 位	5 位	5 位	6 位	所有 MIPS 指令都是 32 位
R 型	op	rs	rt	rd	shamt	funct	算术指令型
I 型	op	rs	rt	地址/立即数			传输、分支和立即数型
J 型	op	目标地址					跳转指令型

4、分支指令细节：

beq:

PC: beq r1, r2, L1

PC+4：...

若r1=r2，则跳转到PC+4+L1×4的地方执行指令

```
Loop:sll    $t1,$s3,2
     add $t1,$t1,$s6
     lw  $t0,0($t1)
     bne $t0,$s5,Exit
     addi $s3,$s3,1
     j   Loop
Exit:
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	……					

8不等于21，所以跳到80016+2*4=80024的地方

J：

PC:J Exit

所跳转的地方为：取PC高4位作为跳转地址的高4为，取Exit×4作为跳转地址的其它位

```
Loop:sll    $t1,$s3,2
     add $t1,$t1,$s6
     lw  $t0,0($t1)
     bne $t0,$s5,Exit
     addi $s3,$s3,1
     j   Loop
Exit:
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	……					

取PC高4位为0000作为跳转地址的高4位，取20000*4作为地址的剩余位
合成后地址为80000

5、过程调用：

过程可以近似理解为函数

寄存器$a0~$a3为参数寄存器，用于调用函数时的参数准备

寄存器$v0~$v1为返回值寄存器，用于存放函数的返回值

寄存器$ra用于存放返回地址，调用函数前需将PC+4存到$ra中，用于函数调用完后返回

寄存器$sp为栈指针寄存器，在调用函数前，需要将通用寄存器的值保存到栈中，调用函数后需根据栈恢复通用寄存

叶过程：不调用其它过程

非叶过程：嵌套调用其它过程

过程调用步骤：

1将参数放到$a0~$a3中，多余的参数放到内存中

2进入到函数中

3（函数中）若有用到通用寄存器，则先将栈指针寄存器$sp自减，开出栈空间，然后将通用寄存器的值保存到栈中

4（函数中）执行任务，若有返回值则将返回值存在$v0中

5（函数中）将栈中保存的值恢复到通用寄存器中，然后$sp自增，恢复栈空间

6退出函数，依据$ra回到调用函数时的下一个地址

以sort过程的MIPS版本举例：

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)
循环体外		move	$s0, $zero # i = 0
	for1tst:	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
		ja	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

6、同步：

```
try:add   $t0, $zero, $s4    ;copy exchange value
    ll    $t1, 0($s1)        ;load linked
    sc    $t0, 0($s1)        ;store conditional
    beq   $t0, $zero, try    ;branch store fails
    add   $s4, $zero, $t1    ;put load value in $s4
```

在指令序列的最后，寄存器 $s4 中的值和 $s1 指向的锁单元的值发生了原子交换。在 ll 和 sc 两条指令之间的任何时候有处理器插入，并修改了该锁单元的值，指令 sc 都会将 $t0 置为 0，引起这段指令序列重新执行。