计系3 期末速通教程

2. MIPS指令

2.1 寄存器使用约定

32 位数据称为一个**字**(word).

寄存器名	寄存器编号	用途	是否调用者保存
$zero	0	常数 0	不可重写
$v0, $v1	$2, 3$	返回值	否
$a0 ~ $a3	$4 \sim 7$	函数传参的前 4 个参数	否
$t0 ~ $t7	$8 \sim 15$	临时变量	否
$s0 ~ $s7	$16 \sim 23$	保存参数	是
$t8, $t9	$24, 25$	临时变量	否
$gp	28	Global Pointer, 静态数据的全局指针	是
$sp	29	Stack Pointer, 堆栈指针	是
$fp	30	Frame Pointer, 帧指针, 保存过程帧的第一个字	是
$ra	31	Return Address, 返回地址	是

2.2 MIPS汇编

2.2.1 算术运算

[**例2.2.1.1**]

```
f = (g + h) - (i + j)
```

将上述C代码翻译为MIPS汇编:

```
add t0, g, h  # t0 = g + h
add t1, i, j  # t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

2.2.2 寄存器操作

[**例2.2.2.1**]

```
f = (g + h) - (i + j)
```

将上述C代码翻译为MIPS汇编, 其中变量 f, g, h, i, j 分别在寄存器 $s0 ~ $s4 中.

```
add $t0, $s1, $s2  # t0 = g + h
add $t1, $s3, $s4  # t1 = i + j
sub $s0, $t0, $t1  # f = t0 - t1
```

2.2.3 内存操作

内存用Byte表示, 每个地址指向一个 8 位的字节Byte.

字在内存中对齐, 地址是 4 的倍数.

MIPS用大端法存储, 即权重高的字节在本字的四个字节的最低地址处.

[**例2.2.3.1**]

```
g = h + A[8]
```

将上述C代码翻译为MIPS汇编, 其中变量 g, h 分别在 $s1, $s2 中, 数组 $A[]$ 的首地址在 $s3 中.

```
lw $t0, 32($s3)  # 8 * 4 = 32
add $s1, $s2, $t0
```

[**例2.2.3.2**]

```
A[12] = h + A[8]
```

将上述C代码翻译为MIPS汇编, 其中变量 h 在 $s2 中, 数组 $A[]$ 的首地址在 $s3 中.

```
lw $t0, 32($s3)  # 8 * 4 = 32
add $t0, $s2, $t0
sw $t0, 48($s3)  # 12 * 4 = 48
```

2.2.4 立即数操作

立即数用补码表示.

无立即数的减法指令.

```
addi $s3, $s3, 4  # s3 += 4
addi $s2, $s1, -1  # s2 = s1 - 1
```

2.2.5 常数 0

寄存器 $r0 (即 $zero)表示常数 0 , 不可重写.

```
add $t2, $s1, $zero  # t2 = s1
```

2.2.6 位运算

```
and $t0, $t1, $t2  # t0 = t1 & t2
or $t0, $t1, $t2  # t0 = t1 | t2
nor $t0, $t1, $t2  # t0 = t1 nor t2, 或非
```

无按位取反指令 `not` , 但注意到 $a \text{ nor } b = \text{not } (a \text{ or } b)$, 则 `not` 可翻译为:

```
nor $t0, $t1, $zero  # t0 = not t1
```

2.2.7 跳转指令

若条件为真, 则跳转到指定标签执行, 否则继续执行.

```
beq rs, rt, L1  # if (rs == tt) goto L1;
bne rs, rt, L1  # if (rs != tt) goto L1;
```

无条件跳转.

```
j L1  # goto L1;
```

2.2.8 if

MIPS只提供了判断相等和不等的指令.

[**例2.2.8.1**]

```
if (i == j) f = g + h;
else f = g - h;
```

将上述C代码翻译为MIPS汇编, 其中变量 f, g, h, i, j 分别在寄存器 $s0 ~ $s4 中.

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2  # f = g + h
        j Exit
Else:   sub $s0, $s1, $s2  # f = g - h
Exit:   ...
```

set指令当条件为真时将指定寄存器置 1 , 否则置 0 .

```
slt rd, rs, rt  # rd = (rs < rt ? 1 : 0);
slti rt, rs, 常数  # rt = (rs < 常数 ? 1 : 0);
```

`beq` 指令和 `bne` 指令结合其他指令使用可实现其他条件判断.

```
slt $t0, $s1, $s2  # t0 = (s1 < s2 ? 1 : 0);
bne $t0, $zero, L  # if (s1 < s2) goto L;
```

2.2.9 while

[**例2.2.9.1**]

```
while (save[i] == k) i++;
```

将上述 C 代码翻译为 MIPS 汇编, 其中变量 i, k 分别在 `$s3` , `$s5` 中, 数组 $save[]$ 的起始地址在 $s6 中.

```
Loop:   sll $t1, $s3, 2  # t1 = s3 * 4, 即 t1 = 4 * i
        add $t1, $t1, $s6  # t1 = t1 + s6, 即 t1 = save + 4 * i
        lw $t0, 0($t1)  # t0 = M[t1], 即 t0 = save[i]
        bne $t0, $s5, Exit  # if (t0 != s5) goto Exit;
        addi $s3, $s3, 1  # i++
        j Loop
Exit:   ...
```

2.2.10 过程调用

[过程调用]

```
jal L  # goto L
```

(1) `jal` , jump and link, 跳转和链接.

(2) 执行 `jal` 指令后会将 `jal` 指令的下一条指令的地址存到 `$ra` 中, 并跳转到目标地址.

[过程返回]

```
jr $ra  # return
```

(1) 复制 `$ra` 中的值到 PC .

(2) 也可用于运算后的跳转, 如 `switch` .

[例2.2.10.1] 非嵌套过程.

```c
int func(int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

将上述 C 代码翻译为 MIPS 汇编, 其中变量 f 在 `$s0` 中, 返回值在 `$v0` 中.

因 g, h, i, j 是函数的四个参数, 则它们分别在 `$a0` $\sim$ `$a3` 中.

因需用到 `$s0` , 而 `$s0` 是调用者保存的寄存器, 故需先将其值保存在栈帧中再使用.

```
func:   addi $sp, $sp, -4  # 分配 1 个 int 的栈帧
        sw $s0, 0($sp)  # 保存 $s0 的值

        add $t0, $a0, $a1  # t0 = g + h
        add $t1. $a2, $a3  # t1 = i + j
        sub $s0, $t0, $t1  # f = t0 - t1
        add $v0, $s0, $zero  # 返回值

        lw $s0, 0($sp)  # 恢复 $s0 的值
        addi $sp, $sp, 4  $ 回收栈帧

        jr $ra  # return;
```

[**例2.2.10.2**] 嵌套过程.

```c
int fac(int n) {
    if (n < 1) return 1;
    else return n * fac(n - 1);
}
```

将上述 C 代码翻译为 MIPS 汇编.

参数 n 在 `$a0` 中, 返回值在 `$v0` 中.

```
fac:    addi $sp, $sp, -8  # 分配 2 个 int 的栈帧
        sw $ra, 4($sp)  # 保存返回地址
        sw $a0, 0($sp)  # 保存参数 n 的值, 用于后续计算 n * fac(n - 1)

        slti  $t0, $a0, 1  # t0 = (a0 < 1 ? 1 : 0)
        beq $t0, $zero, L1  # if (n >= 1) goto L1;

        addi $v0, $zero, 1  # v0 = 1
        addi $sp, $sp, 8  # 回收栈帧
        jr $ra  # return;

L1:     addi $a0, $a0, -1  # n--;
        jal fac  # goto fac;

        lw $a0, 0($sp)  # 恢复 n 的值
        lw $ra, 4($ra)  # 恢复 $ra 的值
        addi $sp, $sp, 8  # 回收栈帧
        mul $v0, $a0, $v0  # v0 *= n
        jr $ra  # return;
```

其中第二个 `lw` 指令不放在最后一个 `jr` 指令之前是因为访存后需经过几个时钟周期才可以继续流水, 将两个 `lw` 指令放在一起, 是通过调整指令顺序以提高流水效率.

2.2.11 字节/半字操作

```
lb rt, offset(rs)  # 将 M[rs + offset] 的内容符号扩展至 32 位后加载到 rt 中
lh rt, offset(rs)  # 将 M[rs + offset] 的内容符号扩展至 16 位后加载到 rt 中

lbu rt, offset(rs)  # 将 M[rs + offset] 的内容零扩展至 32 位后加载到 rt 中
lhu rt, offset(rs)  # 将 M[rs + offset] 的内容零扩展至 16 位后加载到 rt 中

sb rt, offset(rs)  # 将 rt 符号扩展至 32 位后存到 M[rs + offset] 中
sh rt, offset(rs)  # 将 rt 符号扩展至 16 位后存到 M[rs + offset] 中
```

[**例2.2.11.1**]

```c
void strcpy(char x[], char y[]) {
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0') i++;
}
```

将上述C代码翻译为MIPS汇编, 其中变量 i 在 $s0 中.

```
strcpy: addi $sp, $sp, -4  # 分配 1 个 int 的栈帧
        sw $s0, 0($sp)  # 保存 $s0 的值
        add $s0, $zero, $zero  $ i = 0

L1:     add $t1, $s0, $a1  # t1 = y + i
        lbu $t2, 0($t1)  # t2 = y[i] , 进行零扩展
        add $t3, $s0, $a0  # t3 = x + i
        sb $t2, 0($t3)  # x[i] = t2 , 进行零扩展

        beq $t2, $zero, L2  # if (y[i] == '\0') goto L2;
        addi $s0, $s0, i  # i++;
        j L1

L2:     lw $s0, 0($sp)  # 恢复 $s0 的值
        addi $sp, $sp, 4  # 回收栈帧
        jr $ra  # return;
```

2.2.12 32-bit立即数

大部分常数较小, 用 16 位表示已足够.

可用指令 `lui rt, constant` 取 32 位立即数.

汇编的伪指令中允许直接使用 32 位立即数, 汇编器利用 `lui` 指令和 $at 寄存器用两条指令完成.

[**例2.2.12.1**] 将 32 位立即数 0000 0000 0011 1101 0000 1001 0000 0000 加载到寄存器 $s0 中.

```
lui $s0, 61  # 高 16 位
ori $s0, $s0, 2304  # 低 16 位
```

2.2.13 伪指令

[**伪指令**] 汇编指令的变种.

(1) 除伪指令外, 大部分汇编指令与机器指令一一对应.

(2) 汇编程序用 $at 作临时寄存器.

[**例2.2.13.1**]

```
move $t0, $t1  # add $t0, $zero, $t1
blt $t0, $t1, L  # slt $at, $t0, $t1
                 # bne $at, $zero, L
```

2.2.14 冒泡排序案例

[**例2.2.14.1**] 冒泡排序.

(1) 变量 tmp 在 $t0 中.

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

对应的汇编:

```
# $t0: tmp
# $a0: v[]
# $a1: k

swap:
    sll $t1, $a1, 2  # t1 = k * 4
    add $t1, $a0, $t1  # t1 = &v[k]
    lw $t0, 0($t1)  # tmp = v[k]
    lw $t2, 4($t1)  # t2 = v[k + 1]
    sw $t2, 0($t1)  # v[k] = v[k + 1]
    sw $t0, 4($t1)  # v[k + 1] = tmp
    jr $ra  # return;
```

(2) 变量 i 在 $s0 中, j 在 $s1 中.

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

过程主体:

```
# s2: v[]
# s3: n
# s0: i
# s1: j

# 移动参数
    move $s2, $a0  # s2 = a0
    move $s3, $a1  # s3 = a1

```

```
# 外层循环
    move $s0, $zero  # i = 0
for1tst:
    slt $t0, $s0, $s3  # t0 = [i < n]
    beq $t0, $zero, exit1  # if (!t0) goto exit1;

# 内层循环
    addi $s1, $s0, -1  # j = i - 1
for2tst:
    slti $t0, $s1, 0  # t0 = [j < 1]
    bne $t0, $zero, exit2  # if (t0) goto exit2;
    sll $t1, $s1, 2  # t1 = j * 4
    add $t2, $s2, $t1  # t2 = &v[j]
    lw $t3, 0($t2)  # t3 = v[j]
    lw $t4, 4($t2)  # t4 = v[j + 1]
    slt $t0, $t4, $t3  # t0 = [v[j + 1] < v[j]]
    beq $t0, $zero, exit2  # if (!t0) goto exit2;

# 为 swap() 准备参数并调用
    move $a0, $s2  # a0 = v
    move $a1, $s1  # a1 = j
    jal swap  # call swap()

# 内层循环
    addi $s1, $s1, -1  # j--
    j for2tst  # goto for2tst;

# 外层循环
    addi $s0, $s0, 1  # i++
    j for1tst  # goto for1tst;
```

过程:

```
sort:
# 分配栈帧, 保存寄存器的值
    addi $sp, $sp, -20  # 分配 5 个 int 的栈帧
    sw $ra, 16($sp)
    sw $s3, 12($sp)
    sw $s2, 8($sp)
    sw $s1, 4($sp)
    sw $s0, 0($sp)

# 过程主体
    ...

# 恢复寄存器的值, 回收栈帧
exit1:
    lw $s0, 0($sp)
    lw $s1, 4($sp)
    lw $s2, 8($sp)
    lw $s3, 12($sp)
    lw $ra, 16($sp)
    addi $sp, $sp, 20

    jr $ra  # return;
```

2.3 MIPS指令

MIPS将每条指令编码为 32 位指令字.

2.3.1 R-型指令

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

字段:

(1) op : opcode, 操作码.

(2) rs : source, 第一个源寄存器编号.

(3) rt : 第二个源寄存器编号.

(4) rd : destination, 目的寄存器编号.

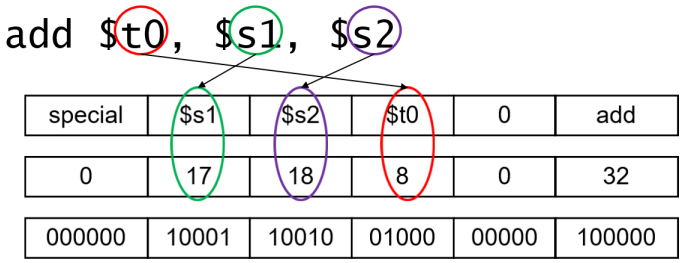
(5) $shamt$: 缩写谐音"杀马特", shift amount, 移位量.

① 逻辑左移sll: 空位补 0 .

② 逻辑右移srl: 空位补 0 .

(6) $funct$: function, 功能码.

[例2.3.1.1]



2.3.2 I-型指令

将R-型指令中的rd、shamt、funct字段合并为constant or address字段.

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

字段:

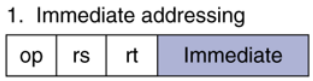
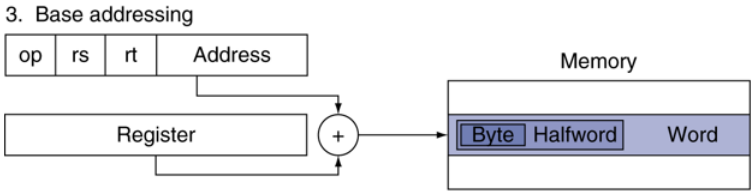
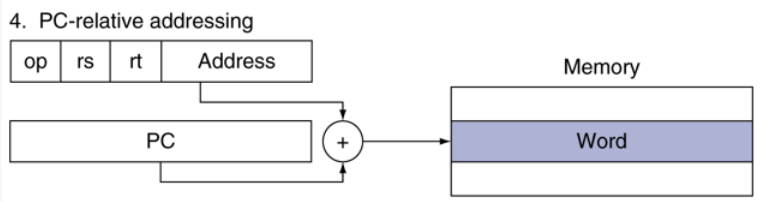
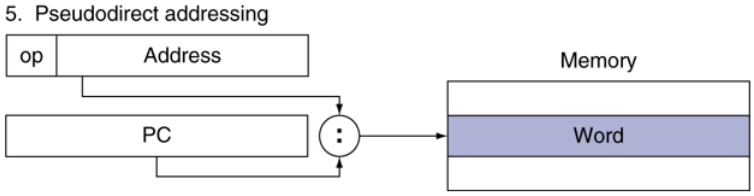
(1) rt : 源寄存器或目的寄存器编号.

(2) 常数的取值范围: $[-2^{15}, 2^{15}-1]$.

(3) 地址 $=rs$ 中的基址 $+$ 常数偏移量.

2.4 寻址

2.4.1 寻址模式总结

寻址模式	示意图
立即数	
寄存器	
基址	
PC相对寻址	
伪地址直接寻址	

[跳转范围]

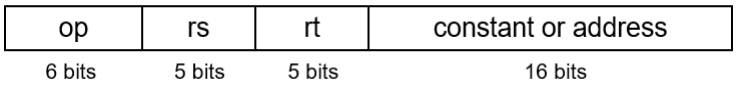
(1) `j` 指令和 `jal` 指令: 2^{28} .

(2) `jr` 指令: 2^{32} .

2.4.2 分支地址

[分支指令]

(1) 分支指令:



(2) 大多数跳转目标离跳出位置较近(向前或向后).

[PC相对寻址]

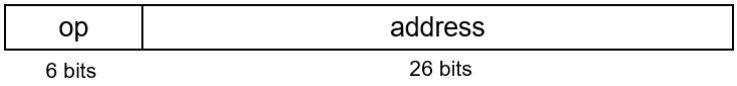
(1) 目标地址 $= \mathrm{PC} + \mathrm{offset} \times 4$.

(2) `PC` 的增量是 4 的倍数.

2.4.3 跳转地址

[跳转指令]

(1) `j` 指令和 `jal` 指令的目标地址可为代码段的任一位置.



(2) 目标地址 $= PC[31:28] : (\mathrm{address} \times 4)$, 即 `PC` 的高 4 位拼上 address 左移两位的结果.

[例2.4.3.1]

```
Loop: sll  $t1, $s3, 2     80000
      add  $t1, $t1, $s6   80004
      lw   $t0, 0($t1)     80008
      bne  $t0, $s5, Exit  80012
      addi $s3, $s3, 1     80016
      j    Loop            80020
Exit: …                    80024
```

0	0	19	9	4	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

2.4.4 远程分支

若跳转对象的地址无法用 16 位偏移地址表示, 则汇编将重写代码.

[例2.4.4.1] 将短跳转(2^{16} 范围)变为长跳转(2^{26}范围).

```
    beq $s0, $s1, L1  # 若 L1 的偏移地址无法用 16 位立即数表示

# 则重写为
    bne $s0, $s1, L2
    j L1
L2:
    ...
```
