# 第二章学习笔记

**设计原则1**：简单源于规整

对于以个给定的功能，使用java，C，MIPS汇编语言写代码，那么他们三种语言写出来的代码行数最多?

排序：MIPS， C， Java。java完成编译延迟于C。

## 3. 计算机硬件的操作数

与高级程序不同，MIPS的运算操作数必须要来自寄存器。寄存器的大小一般为32位，也就是1字，字是计算机中的基本访问单位。约定寄存器有以下表示：

- t0, t1, ... , t9 用于存放临时变量
- s0, s1, ..., s7 表示需要保存的变量

**设计原则2**：越小越快。大量的寄存器会使得时钟周期变长。因此寄存器是一个很小的空间，在这个空间里面，把算术运算放入进行，于是就速度很快。

MIPS用一个`$`符号来表示寄存器。那么`$s0`表示的是s0寄存器。

**例1**：写出`f = (g+h)-(i+j)`的mips指令，其中fghij一次放在$s0 ~ $s4.

```
add $t0 $s1 $s2 #reg $t0 contains g+h
add $s3 $s3 $s4
sub $s0 $t0 $t1
```

### 3. 1 存储器操作数

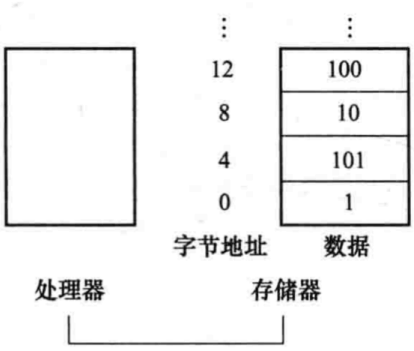
我们已经知道， 寄存器里面放的是运算中所需要的数据，寄存器的容量很小，不是所有的数据都需要放进去。存储器有数十亿的数据，所以复杂的数据结构等需要大容量的数据都是放在**存储器**里的。

取数load指令：取指令的格式就是操作码后面跟着目标寄存器，在后面是访问存储器的常数和寄存器。常数和第二个寄存器相加得到了存储器中的数据地址，将数据读取，写入到寄存器里。这样的过程就是load word，MIPS指令的助记符为lw。

**例2**：写出g = h + A[8], g和h分别在$s1, $s2

```
lw $t0 8($s2) #temorary reg $t0 gets A[8]
add $s1 $t0 $s2 #g = h + A[8]
```

实际情况下， 很多体系结构都是按照字节编址的，也就是说一个字的地址必须和它所包括的4字节中的某个地址相匹配，且连续字的地址相差4。比如下面这张图所展示的字节地址。




> 计算机的寻址方式：大小端问题
>
> big end 大端：从最左编开始读； little end是从最右边开始读起。
>
> 简而言之，big end 就是高内存地址放整数的地位，低内存地址存放整数的高位。这里有一个例子：对于0x12345678，大小段的存放方式如下：

| 地址 | big end | small end |
| --- | --- | --- |
| 低 | 78 | 12 |
| | 56 | 34 |
| | 34 | 56 |
| 高 | 12 | 78 |

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

```
lw $t0 32(s0)
add $t1 $t0 $s2
sw $s1 48($t1)
```

编译器尽量将最常用的变量都保存在寄存器中，其他的变量保存在存储器中，方法是使用存数，或者取数。

### 3.2 常数或立即数操作数

一般，常数都放在存储器中，但是对常数的使用实在是太频繁了，于是采用了一个新的方法就是加立即数（immediate），它在指令中，操作的时候，他直接读取到alu里面做计算。

非常慢：由于程序是通过计算机语言实现的，而指令集体系结构具有惯性，因此寄存器数目的增长要与新指令集的可行性保持一致。

数组的基地址一般放在寄存器里。

### 3.3 有符号数和无符号数

overflow溢出：操作结果不能被最右端的硬件表示。

MIPS 字段：

opcode ～rs ～rt ～constant

```
# i in $s3, k in $s5, addr in $s6
Loop: sll  $t0, $s3, 2 ## i * 4
    add  $t0, $t0, $s6
    lw   $t2, 0($t0)
    bne  $t2, $s5, Else
    addi $s3, $s3, 1
Else:
```

练习：

```
if(i >= j){
  i = i + 1;
}else {
  j = j + 1;
}
```

```
# i in $s3, j in $s5
slt $t0, $s3, $s5 ##if i < j, t0 = 1
bne $t0, zero, Else
addi $s3, $s3, 1
J Exit

Else:
  addi $s5, $s5, 1
Exit:
```

| 保留 | 不保留 |
| --- | --- |
| 保存寄存器：$s0 ~ $s7 | 临时寄存器：$t0 ~ $t9 |
| 栈指针寄存器：$sp | 参数寄存器：$a0 ~ $a3 |
| 返回地址寄存器：$ra | 返回值寄存器：$v0 ~ $v1 |
| 栈指针以上的栈 | 栈指针以下的栈 |

图 2-11　过程调用时，保留和不保留的内容。如果软件依赖于下面将讨论的帧指针寄存器或者全局指针寄存器，那么它们也需要保留

固定值0存放在寄存器`$zero`里.

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

这涉及到了递归调用。我们可以用参数寄存器（a0～a3）或者临时寄存器（t0～t9）进行压栈。被调用者将返回地址寄存器ra和被调用者食用的保存寄存器都压栈，栈指针随着寄存器个数不断进行调整。返回时，寄存器恢复，sp指针也复位。我们可以通过写上面这段代码的汇编语言来进行理解：

```
fact:
    addi $sp, $sp, -8 ##save space for return val and param
    lw   $ra, 4($sp) #load return address to ra
    lw   $a0, 0($sp) #load arg n to a0
    slti $t0, $a0, 1 #judge if n < 1, t0 = 1 if true
    beq  $t0, 0, L1
    addi $a0, $a0, -1#continue i n >= 1
    jal  fact
    ##after fact is finished
    lw   $a0, 0($sp)
    lw   $ra, 4($sp)
    addi $sp, $sp, 8
    mul  $v0, $a0, $v0
    jr   $ra

L1:
    addi $v0, 1, 0
    addi $sp, $sp, 8
    j    $ra
```

*sp*指向堆栈可用地址，申请栈的时候，向下申请。*fp*是指向该帧的第一个字，一般是保存参数寄存器。寄存器还是有很多讲究的，详细可以看下面的这张表，不同的字母代表的含义也有所不同。这种约定也是**加速大概率事件**的另外一个例子。

| 名称 | 寄存器号 | 用途 | 调用时是否保存 |
| --- | --- | --- | --- |
| $zero | 0 | 常数0 | 不适用 |
| $v0 ~ $v1 | 2 ~ 3 | 计算结果和表达式求值 | 否 |
| $a0 ~ $a3 | 4 ~ 7 | 参数 | 否 |
| $t0 ~ $t7 | 8 ~ 15 | 临时变量 | 否 |
| $s0 ~ $s7 | 16 ~ 23 | 保存的寄存器 | 是 |
| $t8 ~ $t9 | 24 ~ 25 | 更多临时变量 | 否 |
| $gp | 28 | 全局指针 | 是 |
| $sp | 29 | 栈指针 | 是 |
| $fp | 30 | 帧指针 | 是 |
| $ra | 31 | 返回地址 | 是 |

图 2-14　MIPS 寄存器约定。称为 $at 的寄存器 1 被汇编器所保留（见 2.12 节），称为 $k0 ~ $k1 的寄存器 26 ~ 27 被操作系统所保留。关于这点也可见 MIPS 参考数据卡的第 2 列

为了管理栈，我们设置了两个指针：sp，fp。fp记录的是栈最高的部分的地址。在函数执行过程中所需保存的局部变量、参数、返回地址调用的过程叫做过程帧。引入$fp可以达到快速恢复栈的目的，避免了内存泄露的问题。

**跳转指令**：区别`jal`和`jr`

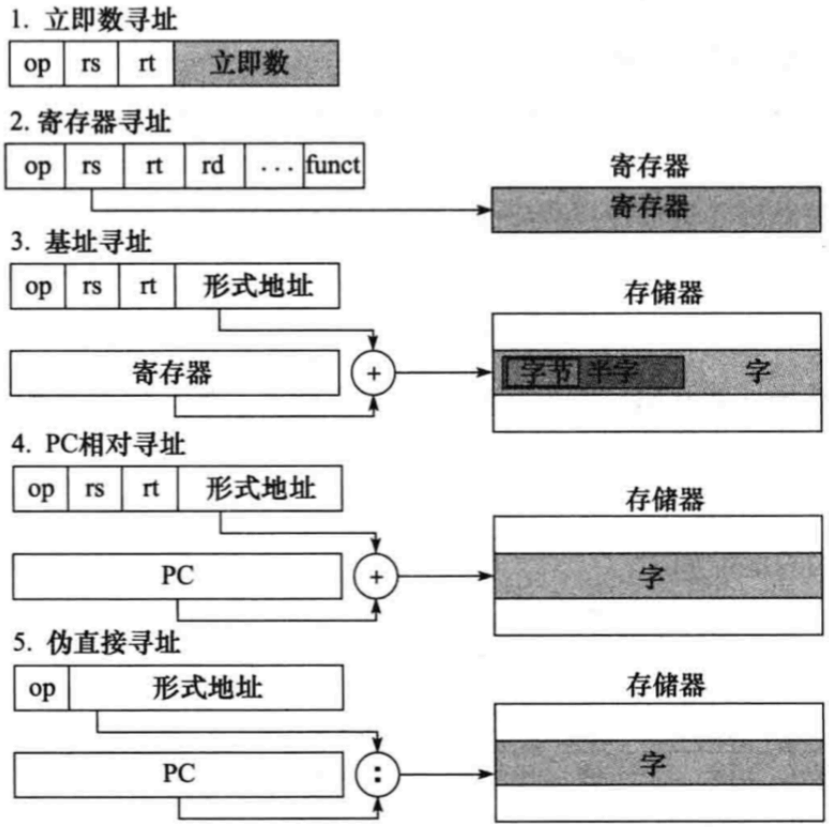
jr：jump register 寄存器跳转，表示无条件跳转到寄存器所指定的地址。

函数调用的基本步骤：

> 将函数相关的参数放入寄存器中(load 指令或者是逻辑运算指令)
>
> 将寄存器的控制权交给函数相关的进程。
>
> 申请，并获得存储空间（堆栈）
>
> 执行函数中的指令
>
> 将返回的结果放回寄存器中
>
> 将结果返回到调用的地址中

**4. MIPS中的32为立即数和寻址**

**MIPS寻址模式总结**




寻址方式分为**数据寻址**和**指令寻址**两种，前者包括了立即数寻址、寄存器寻址和基址寻址，后者包括了pc相对寻址和伪直接寻址。

- 立即数寻址：操作数是位于指令自身中的常数
- 寄存器寻址：操作数是寄存器 (jal $ra)
- 基址寻址 / 偏移寻址：操作数在内存中，其地址是指令中基址寄存器和常数的和。（比如对于数组的访问）
- pc相对寻址：地址是pc和指令中常数的和
- 伪直接寻址：跳转地址由指令中26位字段和pc高位拼接得到。（如，无条件跳转是pc的高四位与addr左移两位拼接的结果，直接赋值跳转jar改变了pc直接跳转）

**MIPS中指令的格式**

| 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 |

图 2-20　MIPS 指令的格式

## 并行与指令：同步

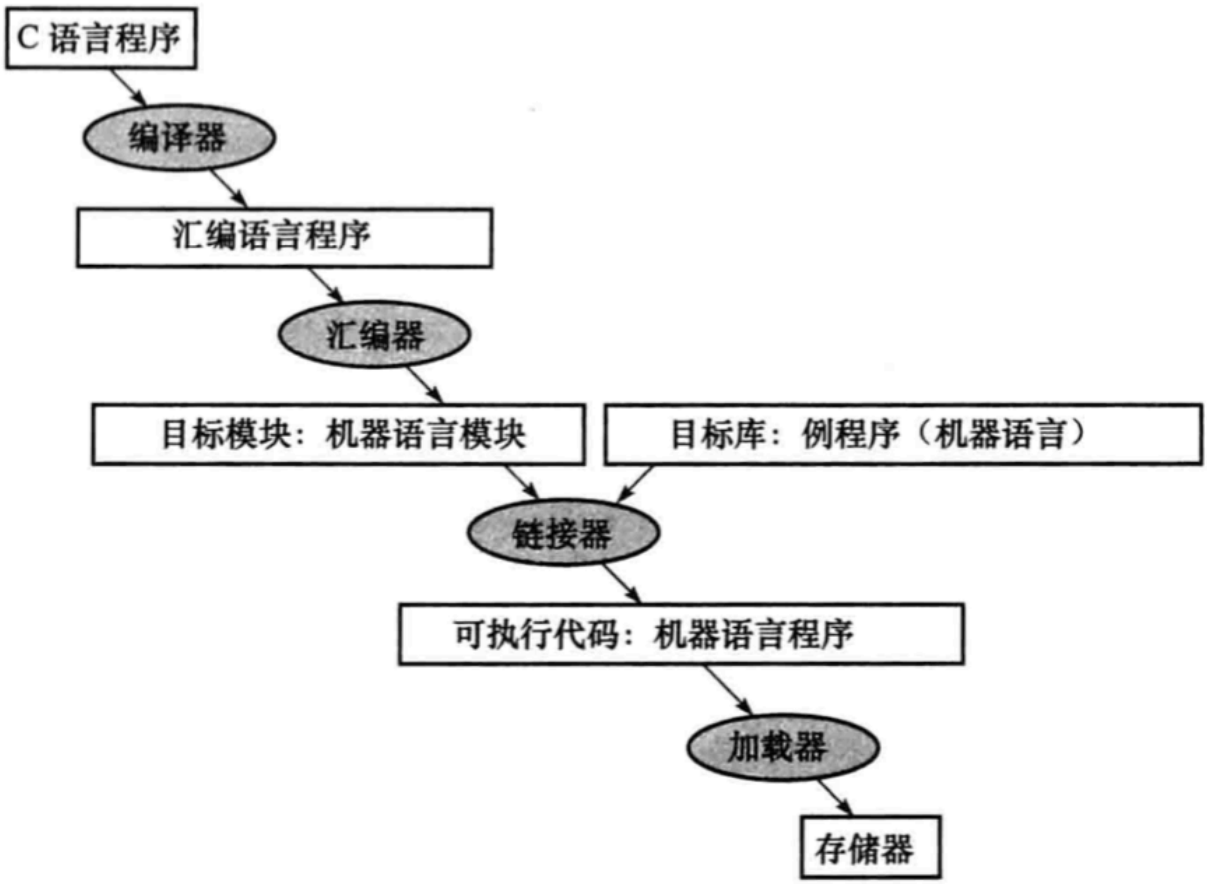
加锁、解锁，互斥。

在 MIPS 处理器中这一指令对包括一条叫作链接取数（load linked）的特殊取数指令和一条叫作条件存数（store conditional）的特殊存数指令。我们顺序地使用这两条指令：如果链接取数指令所指定的锁单元的内容在相同地址的条件存数指令执行前已被改变，那么条件存数指令就执行失败。我们定义条件存数指令完成以下功能：保存寄存器的值，并且如果执行成功则将寄存器的值修改为 1，如果失败则修改为 0。因为链接取数指令返回锁单元的原始值，条件存数指令执行成功的时候才返回 1，下面的指令序列实现了存储器单元的原子交换。存储器单元的地址由 $s1 中的值指出。

```
again:
addi $t0, $zero, 1 # t0 = s4
ll   $t0, 0($s1)
sc   $t0, 0($s1)
beq  $t0, $zero, again
add  $s4, $zero, $t1 #put load value in s4
```

## 翻译、执行程序

流程如下图所示：



## 小测验答案

**2.2** MIPS，C，Java

**2.3** 2. 非常慢

**2.4** 2. $-8_{10}$

**2.5** 4. `sub $t2, $t0, $t1`

**2.6** 都可以。将“逻辑与”和全“1”的掩码一起使用会导致除了想要的区域之外，都变成0。正确的左移位操作将左边的位数都移走。合适的右移将一个字最右边的区域都移走，将0留在字中。注意到“逻辑与”操作会保留原始的值，移位操作对将需要的区域移动到字的最右边。

**2.7** I. 全对，II. 1。

**2.8** 两个都正确。

**2.9** I. 1和2，II. 3。

**2.10** I. 4. + −128K，II. 6. 一个256M的块，III. 4. `sll`。

**2.11** 两个都正确。

**2.12** 4. 与机器无关。