MIPS 汇编语言简要介绍

一、数据类型和文法、寄存器

1、数据类型和文法

（1）数据类型：字节，byte 占用 8bits；半字，halfword 占用 2 bytes（16bits）；字，word 占用 4bytes（32bits）；双字，dword 占用 8bytes（64bits）；

（2）一个字符需要 1 个 Byte 的空间；

（3）一个整数需要 1 个 Word（4 Bytes）的空间；

（4）MIPS 结构的每条指令长度都是 32bits，一个字的空间。

2、寄存器

（1）MIPS 体系架构有 32 个通用寄存器。在汇编程序中，可以用编号 0 到 31 来表示；

（2）也可以用寄存器的名字来进行表示， 例如： $v0、$v1、$s0、$sp、$ra….

（3）有两个特殊的寄存器 Lo, Hi, 用来保存乘法/除法的运算结果；此两个寄存器不能直接寻址，只能用特殊的指令：mfhi 和 mflo 来存取其中的内容。（含义：mfhi = move from Hi, mflo = Move from Low.）

（4）堆栈（Stack）的增长方向是: 从内存的高地址方向, 向低地址方向；

表格 1：寄存器的编号名称及分类

寄存器编号	寄存器名称	寄存器描述
0	$zero	第 0 号寄存器，其值始终为 0
1	$at	(Assembler Temporary) 是 Assembler 保留的寄存器
2、3	$v0，$v1	(values)保存表达式或函数返回的结果
4-7	$a0-$a3	(arguments) 作为函数的前四个入参。在子函数调用的过程中不 会被保留。
8-15	$t0-$t7	(temporaries) Caller saved if needed. Subroutines can use without saving.供汇编程序使用的临时寄存器。在子函数调用的过程中不会被保留。
16-23	$s0-$s7	(saved values) - Caller saved. A subroutine using one of these must save original and restore it before exiting. 在子函数调用的过程中会被保留。
24、25	$t8，$t9	(temporaries) Caller saved if needed. Subroutines can use without saving.供汇编程序使用的临时寄存器。在子函数调用的过程中不会被保留。这是对 $t0 - $t7 的补充。
26、27	$k0，$k1	保留，仅供中断(interrupt/trap)处理函数使用
28	$gp	global pointer. 全局指针。Points to the middle of the 64K block of memory in the static data segment.指向固态数据块内存的 64K 的块的中间。
29	$sp	stack pointer 堆栈指针， 指向堆栈的栈顶。
30	$s8/$fp	saved value / frame pointer 保存的值/帧指针 其中的值在函数调用的过程中会被保留
31	$ra	return address 返回地址

二、汇编语言程序结构框架

汇编源程序代码本质上是文本文件。由数据声明、代码段两部分组成。汇编程序文件应该以.s 或.asm 为后缀，在 Spim 模拟器中进行模拟。

1、数据声明部分

在源代码中，数据声明部分以“.data”开始。声明了在代码中使用的变量的名字。同时，也在主存（RAM）中创建了对应的空间。

2、程序代码部分

在源代码中，程序代码部分以 “.text”开始。这部分包含了由指令构成的程序功能代码。代码以 main: 函数开始。main 的结束点应该调用 exit system call，参见后文有关“表 2：系统调用的功能”。

3、程序的注释部分

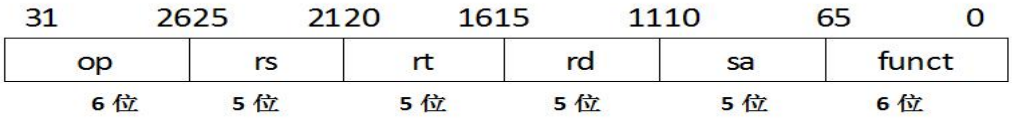
使用“#”符号进行注释。每行以“#”引导的部分都被视作注释。

程序结构框架，如：

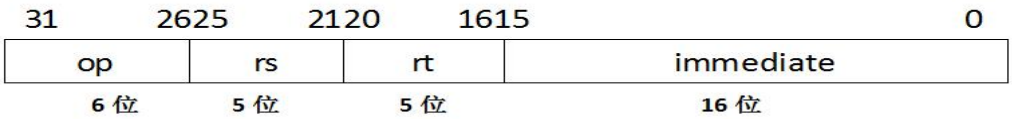
```
.data          # 数据声明部分
               # … 变量类型：
               # 字符串：
               #    .ascii   string
               #    .asciiz   string ；以空字符 null 结束
               # 字类型：
               #    .word    w1,w2,…   ;32 位，4 个字节
               # 半字类型：
               #    .half    h1,h2,…      ; 16 位，2 个字节
               # 字节类型：
               #    .byte    b1,b2,…       ; 8 位，1 个字节
               # 浮点数类型：
               #    .float    f1,f2,…        ; 32 位，4 个字节
               # 双精度浮点数：
               #    .double    d1,d2,…     ; 64 位，8 个字节
               # 空格符：
               #    .space    n              ; 8 位，1 个字节。n 个字节空间
.text          # 代码声明部分
.globl main    # 定义 main 为全程量
main:          #主程序名称 main，以下为程序代码部分
               # 编写的 MIPS 汇编语言程序是 MIPS 指令和伪指令的组合。
               # 伪指令（pseudo instructions）是为了编程方便而对指令集进行的扩展。
# 程序结束
```

四、MIPS 指令的三种格式：

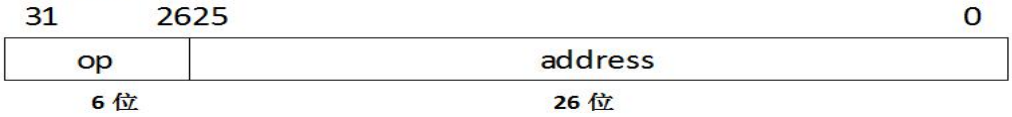
R 类型:



I 类型:



J 类型:



其中，

op：为操作码；

rs：为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt：为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

immediate：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Laod）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

五、关于 MIPS 伪指令与 MIPS 指令的转换关系

表 1：一些 MIPS 伪指令与指令的转换关系举例

MIPS 伪指令	功能描述	实现前伪指令功能所需相应的 MIPS 指令及方法	功能描述
move $t2,$t4	$t2<-$t4	addu $t2,$zero,$t4	$t2<-$zero + $t4
b imm16	无条件转移，pc<-pc+4+sign-extend(imm16)	beq $zero,$zero,imm16	$zero=$zero，pc<-pc+4+sign-extend(imm16)
li $t1,10	$t1<-10	ori $t1,$zero,10	$t1<-$zero | 10
la $t2,0x1000056c	$t2<-0x1000056c	lui $at,0x1000	0x1000=4096，0x10000=65536 $at <- 4096*65536；将 16 位立即数放到目标寄存器($at)高 16 位，目标寄存器的低 16 位填 0
		ori $t2,$at,0x056c	$t2<-$at | 0x056c，$t2<-重新合并该数
beqz $t1,imm16	If $t1=0	beq $t1,$zero,imm16	If $t1=$zero

	pc<-pc+4+sign-extend(imm16) else pc<-pc+4		pc<-pc+4+sign-extend(imm16) else pc<-pc+4
bnez $t1,imm16	If $t1!=0 pc<-pc+4+sign-extend(imm16) else pc<-pc+4	bne $t1,$zero,imm16	If $t1!=$zero pc<-pc+4+sign-extend(imm16) else pc<-pc+4
bleu $t1,100,imm16	If $t1<=100 pc<-pc+4+sign-extend(imm16) else pc<-pc+4	ori $at,$zero,100	$at<-$zero|100，或运算
		beq $t1,$at,imm16	If $t1=$at （注：$t1=100） pc<-pc+4+sign-extend(imm16) else pc<-pc+4
		sltu $at,$at,$t1	If $at<$t1 $at=1 else $at=0, 无符号数
		beq $at,$zero,imm16	If $at=$zero （注：$t1<100） pc<-pc+4+sign-extend(imm16) else pc<-pc+4

注：

（1）imm16 实际上是偏移的指令条数，即从 pc+4 地址的指令开始考虑偏移的指令条数，符号补码数，或者标签 label。

（2）表 1 只是列举几条伪指令与指令的转换关系。

（3）表 3：MIPS32 指令集，在本文档后面，可供参考。

六、MIPS 汇编语言程序大概包括几部分内容：

PartI: 数据的声明

Part II：数据的装载和保存（Load/Store 指令）

Part III：寻址

Part IV：算术运算指令：Arithmetic Instructions

Part V：程序控制指令：Control Instructions

Part VI：系统调用和 I/O 操作（SPIM 仿真）

1、PartI：数据的声明

格式： **name: .storage_type value(s)**

创建一个以 name 为变量名称，values 通常为初始值，storage_type 代表存储类型。

注意：变量名后要跟一个“:”冒号。

例子：

```
.data       0x10010100                  # 在 .data 后面还可以指定数据段的开始地址，如
                                        # 0x10010100 即为数据从 0x10010100 地址开始存放
        var1:     .word   3             # 定义为字类型，初始值为 3
      array1:   .byte   'a','b'         # 定义为字节类型，初始值为两个字符，“a”和“b”
      array2:    .space 30              # 给变量分配 30 个字节空间
      string1:  .asciiz "hello world.\n"   # 定义为字符串，初始值为"hello world.”，字符串以
                                           # null 结尾
```

2、Part II：数据的装载和保存（Load/Store 指令）

主存（RAM）的存取 access 只能用 load / store 指令来完成。所有其他的指令都使用的是寄存器作为操作数。

i. load 指令:

```
lw    register_destination, RAM_source # 从内存地址 RAM_source 的单元中读取一个字数据
                                       # 保存到寄存器 destination_register 中
lb    register_destination, RAM_source  # 从内存地 RAM_source 的单元中读取一个字节数据
                                       # 保存到寄存器 destination_register 中
li    register_destination, value       # 将立即数 value 保存到寄存器 destination register 中
```

ii. store 指令

```
sw    register_source, RAM_destination   # 将寄存器 register_source 中的字数据保存到内存
                                         # 地址为 RAM_destination 的字单元中
sb    register_source, RAM_destination   # 将寄存器 register_source 中的低 8 位数据保存到
                                         # 内存地址为 RAM_destination 的字节单元中
```

程序例子：

```
.data
      var1:    .word   22       # 字类型变量 var1，初值 22
.text
.globl main
main:
    lw   $t0, var1          # $t0 ← [ var1]，[var1]=22；[var1]表示为 var1 地址单元内容，如 22
    li   $t1, 5             # $t1 ← 5
    sw   $t1, 0($t0)        # [$t0+0] ← $t1 ，存入内存单元中
    li   $t1, 13            # $t1 ← 13
    sw   $t1, 4($t0)        # [$t0+4] ← $t1
    li   $t1, -7            # $t1 ← -7
    sw   $t1, 8($t0)        # [$t0+8] ← $t1

    Li   $v0,10             # 退出
    syscall
# done
```

3、Part III：寻址 ：

MIPS 系统结构只能用 load/store 相关指令来实现寻址操作，包含 3 中寻址方式：装载地址：load address，相当于直接寻址，把数据地址直接载入寄存器。

间接寻址，indirect addressing，间接寻址，把寄存器内容作为地址。基线寻址/索引寻址，based or indexed addressing，相对寻址，利用补偿值(offset)寻址。

直接寻址/装载地址：load address:

```
la   $t0, var1
```

把 var1 在主存（RAM）中的地址拷贝到寄存器 t0 中。var1 也可以是程序中定义的一个子程序标签的地址。

间接寻址：indirect addressing:

```
lw   $t2, 0($t0)
```

主存中有一个字的地址存在 t0 中，按这个地址找到那个字数，把字数存到寄存器 t2 中。

```
sw   $t2, 0($t0)
```

把 t2 中的字数存入 t0 中的地址指向的主存单元中。

基线寻址/索引寻址：based or indexed addressing:

```
lw   $t2, 4($t0)        # 偏移量：4
```

把 t0 中地址+4 所得的地址所对应的主存中的字数载入寄存器 t2 中，4 为包含在 t0 中的地址的偏移量。

```
sw   $t2, -12($t0)     # 偏移量：-12
```

把 t2 中的内容存入 t0 中的地址-12 所得的地址所对应的主存中，存入一个字数，占用 4 字节，消耗 4 个内存号，可见，地址偏移量可以是负值。

注意：基线寻址在以下场合特别有用：

1、数组：从基址出发，通过使用偏移量，存取数组元素。

2、堆栈：利用从堆栈指针或者框架指针的偏移量来存取元素。

例子：

```
.data
    array1:   .space 12      # 定义 12 个字节的存储单元
.text
.globl main
main:
    la   $t0, array1         # $t0←array1 地址
    li   $t1, 5              # $t1 ←5
    sw   $t1,0 ($t0)         # [array1+0]←$t1 ，存入内存单元中
    li   $t1, 13             # $t1← 13
    sw   $t1, 4($t0)         # [array1+4]←$t1
    li   $t1, -7             # $t1 ← -7
    sw   $t1, 8($t0)         # [array1+8]←$t1

    Li   $v0,10              # 退出
    syscall
#done
```

4、Part IV 算术运算指令：Arithmetic Instructions

（1）算数运算指令的所有操作数都是寄存器或寄存器与立即数，不能直接使用 RAM 地址或间接寻址。

（2）操作数的大小都为 Word （4-Byte）。（下表都是 MIPS 指令）

指令	功能	说明
add $t0,$t1,$t2	$t0 ← $t1 + $t2	其中，rs=$t1，rt=$t2，rd=$t0（带符号数）
sub $t2,$t3,$t4	$t2← $t3 - $t4	其中，rs=$t3，rt=$t4，rd=$t2
addi $t2,$t3, 5	$t2 ← $t3 + 5	其中，rs=$t3，rt=$t2，immediate=5
addu $t1,$t6,$t7	$t1 ← $t6 + $t7	其中，rs=$t6，rt=$t7，rd=$t1（不带符号数）
addiu $t1,$t6,5	$t1 ←$t6 + 5	其中，rs=$t6，rt=$t1，immediate=5
subu $t1,$t6,$t7	$t1 ← $t6 - $t7	其中，rs=$t6，rt=$t7，rd=$t1
subiu $t1,$t6,5	$t1 ←$t6 - 5	其中，rs=$t6，rt=$t1，immediate=5
mult $t3,$t4	(Hi,Lo) ← $t3 * $t4	Hi<-乘积高于 32 位，Lo<-乘积 32 位以内， 其中，rs=$t3，rt=$t4；读取 Hi、Lo 的方法，如

		mfhi $t6; $t6<-Hi，其中 rd=$t6 mflo $t5; $t5<-Lo，其中 rd=$t5
div $t5,$t6	Lo = $t5 / $t6 Hi = $t5 mod $t6	Lo 为商的整数部分，Hi 为余数， 其中，rs=$t5，rt=$t6；读取 Hi、Lo 的方法，如 mfhi $t7; $t7<-Hi，其中 rd=$t7 mflo $t8; $t8<-Lo，其中 rd=$t8

5、**Part V 程序控制指令：Control Instructions**

（1）分支指令（Branches）

条件分支的比较机制已经内建在指令中：（表中黑颜色指令为 MIPS 指令，蓝色的为伪指令）

指令	功能	说明
beq $t0,$t1,imm16	if $t0 = $t1 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0，rt=$t1	imm16 实际上是偏移的指令条数，即从 pc+4 地址的指令开始考虑偏移的指令条数，符号补码数，或者标签 label。
bne $t0,$t1,imm16	if $t0 = $t1 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0，rt=$t1	
bgez $t0,imm16	if $t0 >= 0 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0	
bgtz $t0,imm16	if $t0 > 0 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0	
blez $t0,imm16	if $t0 <= 0 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0	
bltz $t0,imm16	if $t0 < 0 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0	
b imm16	无条件转移， pc<-pc+4+sign-extend(imm16)	
ble $t0,$t1,imm16	if $t0 <= $t1 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0，rt=$t1	
bgt $t0,$t1,imm16	if $t0 >$t1 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0，rt=$t1	
bge $t0,$t1,imm16	if $t0 >=$t1 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0，rt=$t1	
beqz $t0,imm16	if $t0 = 0 pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0	
bnez $t0,imm16	if $t0 != 0	

	pc<-pc+4+sign-extend(imm16) else pc<-pc+4，其中 rs=$t0	
bgtzal $t0,imm16 有些模拟器没提供该指令	if $t0 > 0 $ra<-pc +4 ，pc<-pc+4+sign-extend(imm16)，其中 rs=$t0	
bltzal $t0,imm16 同上	if $t0 < 0 $ra<-pc +4，pc<-pc+4+sign-extend(imm16)，其中 rs=$t0	

（2）传送指令

move $t0,$t1 # $t0<-$zero+$t1，其中，rd=$t0，rt=$t1；

（3）跳转指令(Jumps)

J address # pc<-address，程序无条件跳转到地址为 address 的指令执行

jr $t2 # pc<-$t2，程序无条件跳转到寄存器$t3 的内容为地址的指令执行，其中 rs=$t2

（4）子程序调用指令

子程序调用指令的实质是跳转并链接（Jump and Link），它把当前程序计数器的值保留到 $ra 中，以备跳回。

跳转到子程序：

jal sub_label # $ra<-pc+4，保存返回地址；pc<-sub_label，程序调用入口地址为 # sub_label 的子程序执行

从子程序返回：

jr $ra # pc<- $ra，程序返回到调用指令的下条指令执行

返回到$ra 中储存的的返回地址对应的位置，$ra 中的返回地址由 jal 指令保存。注意，返回地址存放在$ra 寄存器中。如果子程序调用了下一级子程序，或者是递归调用，此时需要将返回地址保存在堆栈中，因为每执行一次 jal 指令就会覆盖$ra 中的返回地址。

6、Part VI: 系统调用和 I/O 操作（SPIM 仿真）

系统调用是指调用操作系统的特定子程序。系统调用用来在仿真器的窗口中打印或者读入字符串 string，并可显示程序是否结束。用 syscall 指令进行对系统子程序的调用。本操作首先支持$v0 and $a0-$a1 中的相对值调用以后的返回值（如果存在）会保存在$v0 中。

表 2：系统调用（syscall）的功能

Service	Trap code	Input	Output	Notes
print_int	$v0 = 1	$a0 = integer to print	prints $a0 to standard output	
print_float	$v0 = 2	$f12 = float to print	prints $f12 to standard output	
print_double	$v0 = 3	$f12 = double to print	prints $f12 to standard output	
print_string	$v0 = 4	$a0 = address of first character		prints a character string to standard

				output
read_int	$v0 = 5		integer read from standard input placed in $v0	
read_float	$v0 = 6		float read from standard input placed in $f0	
read_double	$v0 = 7		double read from standard input placed in $f0	
read_string	$v0 = 8	**$a0 = address to place string, $a1 = max string length**	reads standard input into address in $a0	
sbrk	$v0 = 9	$a0 = number of bytes required	$v0= address of allocated memory	Allocates memory from the heap
	heap：是由 malloc 之类函数分配的空间所在地。地址是由低向高增长的。 stack:：是自动分配变量，以及函数调用的时候所使用的一些空间，地址是由高向低减少的。			
exit	$v0 = 10			退出
print_char	$v0 = 11	**$a0 = character (low 8 bits)**		
read_char	$v0 = 12		$v0 = character (no line feed) echoed	
file_open	$v0 = 13	$a0 = full path (zero terminated string with no line feed), $a1 = flags, $a2 = UNIX octal file mode (0644 for rw-r--r--)	$v0 = file descriptor	
file_read	$v0 = 14	$a0 = file descriptor, $a1 = buffer address, $a2 = amount to read in bytes	$v0 = amount of data in buffer from file (-1 = error, 0 = end of file)	
file_write	$v0 = 15	$a0 = file descriptor, $a1 = buffer address, $a2 = amount to	$v0 = amount of data in buffer to file (-1 = error, 0 = end of file)	

		write in bytes		
file_close	**$v0 = 16**	$a0 = file descriptor		

例子 1：打印在$t2 中的整数的值

```
.data
.text
.globl main
main:
    li   $t2,22
    move   $a0, $t2   # $a0 ← $t2
    li   $v0, 1          # $v0 ← 1，打印整数，在屏幕上输出整数 22
    syscall              #  系统调用
```

例子 2：从键盘读取一个整数的值

```
.data
    Int_value:   .space 20   #  定义 20 个字节内存单元，如果输入是整数，则字节空间
                             #  应该为 4 的倍数 20=5*4，一个整数占 4 个字节。
.text
.globl main
main:
    li   $v0, 5              # $v0 ← 5，从键盘输入整数
    syscall                  #  系统调用，从键盘输入的数据保存在$v0 寄存器中
    sw   $v0, int_value     # [int_value] ← $v0，存入内存单元中
```

例子 3：打印字符串

```
.data
    string1   .asciiz "hello world\n"    #  定义字符串变量，建立一个空（null）终止符。
.text
.globl main
main:
    la   $a0, string1   # $a0 ← string1 地址
    li   $v0, 4          # $v0 ← 4，打印字符串，在屏幕上输出字符串
    syscall              #  系统调用，输出字符串
```

例子 4：程序执行结束，退出系统调用。在程序代码结束时，增加以下两条指令。

```
    li   $v0, 10         # $v0 ← 10，退出系统调用
    syscall              #  系统调用
```

下表是 MIPS32 指令集，共 31 条，供参考：

表 3：MIPS32 指令集(共 31 条)

助记符	指令格式						指令示例	示例含义	操作及其解释
Bit #	31..26	25..21	20..16	15..11	10..6	5..0			
R-type	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$=$2+$3	rd <- rs + rt; 其中 rs＝$2，rt=$3， rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+$3	rd <- rs + rt;其中 rs＝$2,rt=$3, rd=$1，(不带符号)
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-$3	rd <- rs - rt; 其中 rs＝$2，rt=$3，rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-$3	rd <- rs - rt；无符号数 其中 rs＝$2，rt=$3，rd=$1
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2 & $3	rd <- rs & rt; 其中 rs＝$2，rt=$3，rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2 | $3	rd <- rs | rt; 其中 rs＝$2，rt=$3，rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2 ^ $3	rd <- rs xor rt；异或 其中 rs＝$2，rt=$3，rd=$1
nor	000000	rs	rt	rd	00000	100111	nor $1,$2,$3	$1=~($2 | $3)	rd <- not(rs | rt)；或非 其中 rs＝$2，rt=$3，rd=$1
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ； 其中 rs＝$2，rt=$3，rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ； 其中 rs＝$2，rt=$3，rd=$1 （不带符号）
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	rd <- rt << shamt；左移 shamt 存放移位的位数，也就是指令中的立即数，其中 rt=$2， rd=$1
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	rd <- rt >> shamt ；右移 其中 rt=$2，rd=$1
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	rd <- rt >> shamt；算数右移

									(arithmetic) 注意符号位保留 其中 rt=$2，rd=$1
sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	rd <- rt << rs；左移 其中 rs＝$3，rt=$2，rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	rd <- rt >> rs；右移 其中 rs＝$3，rt=$2，rd=$1
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	rd <- rt >> rs；算数右移 (arithmetic) 注意符号位保留 其中 rs＝$3，rt=$2，rd=$1
jr	000000	rs	00000	00000	00000	001000	jr $31	goto $31	PC <- rs，PC<-返回地址
I-type	op	rs	rt	immediate			Immediate 为一整数		
addi	001000	rs	rt	immediate			addi $1,$2,100	$1=$2+100	rt <- rs + (sign-extend)immediate; 其中 rt=$1，rs=$2
addiu	001001	rs	rt	immediate			addiu $1,$2,100	$1=$2+100	rt <- rs + (zero-extend)immediate; 其中 rt=$1，rs=$2 (不带符号)
andi	001100	rs	rt	immediate			andi $1,$2,10	$1=$2 & 10	rt <- rs & (zero-extend)immediate; 其中 rt=$1，rs=$2
ori	001101	rs	rt	immediate			ori $1,$2,10	$1=$2 | 10	rt <- rs | (zero-extend)immediate; 其中 rt=$1，rs=$2
xori	001110	rs	rt	immediate			xori $1,$2,10	$1=$2 ^ 10	rt <- rs xor (zero-extend)immediate; 其中 rt=$1，rs=$2
lui	001111	00000	rt	immediate			lui $1,100	$1=100*65536	rt <- immediate*65536；将 16 位立即数放到目标寄存器高 16 位，目标寄存器的 低 16 位填 0
lw	100011	rs	rt	immediate			lw $1,10($2)	$1= memory[$2 +10]	rt <- memory[rs + (sign-extend)immediate] ； rt=$1，rs=$2，immediate=10
sw	101011	rs	rt	immediate			sw $1,10($2)	memory[$2+10] =$1	memory[rs + (sign-extend)immediate] <- rt ； rt=$1，rs=$2，immediate=10

beq	000100	rs	rt	immediate	beq $1,$2,10	if($1==$2) goto PC+4+40	if (rs == rt) PC <- PC+4 + (sign-extend)immediate<<2
bne	000101	rs	rt	immediate	bne $1,$2,10	if($1!=$2) goto PC+4+40	if (rs != rt) PC <- PC+4 + (sign-extend)immediate<<2
slti	001010	rs	rt	immediate	slti $1,$2,10	if($2<10) $1=1 else $1=0	if (rs<(sign-extend)immediate) rt=1 else rt=0 ； 其中 rs＝$2，rt=$1
sltiu	001011	rs	rt	immediate	sltiu $1,$2,10	if($2<10) $1=1 else $1=0	if (rs <(zero-extend)immediate) rt=1 else rt=0; 其中 rs＝$2，rt=$1
J-type	op	address					
j	000010	address			j 10000	goto 10000	PC <- {(PC+4)[31..28],address[27..2],0,0} ; address=10000/4
jal	000011	address			jal 10000	$31<-PC+4; goto 10000	$31<-PC+4; PC <- {(PC+4)[31..28],address[27..2],0,0} ; address=10000/4

ASCII 字符代码表 一

高四位 / 低四位		ASCII非打印控制字符										ASCII 打印字符													
		0000					0001					0010		0011		0100		0101		0110		0111			
		0					1					2		3		4		5		6		7			
		+进制	字符	ctrl	代码	字符解释	+进制	字符	ctrl	代码	字符解释	+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符	ctrl
0000	0	0	BLANK NULL	^@	NUL	空	16	▶	^P	DLE	数据链路转意	32		48	0	64	@	80	P	96	`	112	p	
0001	1	1	☺	^A	SOH	头标开始	17	◀	^Q	DC1	设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q	
0010	2	2	☻	^B	STX	正文开始	18	↕	^R	DC2	设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r	
0011	3	3	♥	^C	ETX	正文结束	19	!!	^S	DC3	设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s	
0100	4	4	♦	^D	EOT	传输结束	20	¶	^T	DC4	设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t	
0101	5	5	♣	^E	ENQ	查询	21	§	^U	NAK	反确认	37	%	53	5	69	E	85	U	101	e	117	u	
0110	6	6	♠	^F	ACK	确认	22	■	^V	SYN	同步空闲	38	&	54	6	70	F	86	V	102	f	118	v	
0111	7	7	•	^G	BEL	震铃	23	↨	^W	ETB	传输块结束	39	'	55	7	71	G	87	W	103	g	119	w	
1000	8	8	◘	^H	BS	退格	24	↑	^X	CAN	取消	40	(	56	8	72	H	88	X	104	h	120	x	
1001	9	9	○	^I	TAB	水平制表符	25	↓	^Y	EM	媒体结束	41	)	57	9	73	I	89	Y	105	i	121	y	
1010	A	10	◙	^J	LF	换行/新行	26	→	^Z	SUB	替换	42	*	58	:	74	J	90	Z	106	j	122	z	
1011	B	11	♂	^K	VT	竖直制表符	27	←	^[	ESC	转意	43	+	59	;	75	K	91	[	107	k	123	{	
1100	C	12	♀	^L	FF	换页/新页	28	∟	^\	FS	文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|	
1101	D	13	♪	^M	CR	回车	29	↔	^]	GS	组分隔符	45	-	61	=	77	M	93	]	109	m	125	}	
1110	E	14	♫	^N	SO	移出	30	▲	^6	RS	记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~	
1111	F	15	☼	^O	SI	移入	31	▼	^-	US	单元分隔符	47	/	63	?	79	O	95	_	111	o	127	Δ	Back space

注：表中的ASCII字符可以用：ALT + "小键盘上的数字键" 输入