MIPS指令集

在 MIPS 汇编语言中，指令一般由一个指令名作为开头，后跟该指令的**操作数**，中间由空格或逗号隔开。指令的操作数的个数一般为 0-3 个，每一个指令都有其固定操作数个数。

操作数

即指令操作所作用的实体，可以是**寄存器、立即数或标签**，每个指令都有其固定的对操作数形式的要求，而标签最终会由汇编器转换为立即数。

立即数

即在指令中设定好的常数，可以直接参与运算，一般长度为 16 位二进制。

标签

用于使程序更简单清晰。标签用于表示一个地址，以供指令来引用。一般用于表示一个数据存取的地址（类似于数组名）、或者一个程序跳转的地址（类似于函数名，或者 C 语言中 goto 的跳转目标）。

一、指令分类

MIPS-C 指令集共包括 55 条指令。从细致的功能角度，其被划分为 9 个子类。

（一）R-R运算

R即Register，寄存器。R-R运算指令的操作数均为寄存器，指令的一般格式如下：

```
OP  rs, rt, rd
```

其中，OP为操作码，**rs、rt、rd分别为源寄存器、目标寄存器、目的寄存器**

(1) **add**

将 rs 和 rt 中的数相加，结果存入 rd 中，考虑溢出

```
add rd, rs, rt
```

(2) **addu**

将 rs 和 rt 中的数相加，结果存入 rd 中，忽略溢出

```
addu rd, rs, rt
```

(3) **sub**

将 rs - rt 的结果存入 rd 中，考虑溢出

```
sub rd, rs, rt
```

(4) **subu**

将 rs - rt 的结果存入 rd 中，忽略溢出

```
subu rd, rs, rt
```

(5) **mult**

乘积低32位存放在LO寄存器，高32位存放在HI寄存器。所有操作数均为有符号数

```
mult rs, rt
```

(6) **multu**

乘积低32位存放在LO寄存器，高32位存放在HI寄存器。所有操作数均为无符号数
(因为 multu 为无符号乘法，所以对其进行 0 扩展 1 位后再进行运算)

```
multu rs, rt
```

(7) **div**

商存放在LO寄存器，余数存放在HI寄存器。所有操作数均为有符号数
(如果 GPR[rt]为 0，则 HI/LO 结果不可预料)

```
div rs, rt
```

(8) **divu**

商存放在LO寄存器，余数存放在HI寄存器。所有操作数均为无符号数
(因为 divu 为无符号除法，所以对其进行 0 扩展 1 位后再进行运算)

```

```

```
divu rs, rt
```

(9) **and**

按位与运算

```
and rd, rs, rt
```

(10) **or**

按位或运算

```
or rd, rs, rt
```

(11) **xor**

按位异或运算

```
xor rd, rs, rt
```

(12) **nor**

按位或非运算

```
nor rd, rs, rt
```

(13) **slt**

有符号数比较，如果 rs < rt，则 rd = 1，否则 rd = 0

```
slt rd, rs, rt
```

(14) **sltu**

无符号数比较，如果 rs < rt，则 rd = 1，否则 rd = 0

```
sltu rd, rs, rt
```

(15) **sll**

逻辑左移，将rt寄存器中的数左移 **shamt** 位，结果存入rd寄存器中

```
sll rd, rt, shamt
```

(16) **sllv**

逻辑可变左移，将rt寄存器中的数左移 **rs** 位，结果存入rd寄存器中
(GPR[rs]的 [31:5] 被忽略，**[4:0]** 决定位移量)

```
sllv rd, rt, rs
```

(17) **srl**

逻辑右移，将rt寄存器中的数右移 **shamt** 位，结果存入rd寄存器中

```
srl rd, rt, shamt
```

(18) **srlv**

逻辑可变右移，将rt寄存器中的数右移 **rs** 位，结果存入rd寄存器中
(GPR[rs]的 [31:5] 被忽略，**[4:0]** 决定位移量)

```
srlv rd, rt, rs
```

(19) **sra**

算术右移，将rt寄存器中的数右移 **shamt** 位，结果存入rd寄存器中

```
sra rd, rt, shamt
```

(20) **srav**

算术可变右移，将rt寄存器中的数右移 **rs** 位，结果存入rd寄存器中
(GPR[rs]的 [31:5] 被忽略，**[4:0]** 决定位移量)

```
srav rd, rt, rs
```

（二）R-I运算

R-I运算指令的操作数均为寄存器和立即数，指令的格式如下：

```
OP  rt, rs, immediate
```

其中，OP为操作码，**rt为目标寄存器**，rs为源寄存器，immediate为立即数。通常情况下，immediate为16位有符号数，由于寄存器为32位，所以有时要对immediate进行**符号扩展**或者**零扩展**。

(21) **addi**

将 rs 和 immediate 相加，结果存入 rt 中，考虑溢出。（立即数符号扩展）

```
addi rt, rs, immediate
```

(22) **addiu**

将 rs 和 immediate 相加，结果存入 rt 中，忽略溢出。（注意，这里立即数也是**符号扩展**）

```
addiu rt, rs, immediate
```

(23) **andi**

按位与运算（立即数零扩展）

```
andi rt, rs, immediate
```

(24) **ori**

按位或运算（立即数零扩展）

```
ori rt, rs, immediate
```

(25) **xori**

按位异或运算（立即数零扩展）

```
xori rt, rs, immediate
```

(26) **slti**

有符号数比较，如果 rs < immediate，则 rt = 1，否则 rt = 0
（rs视为有符号数，立即数符号扩展）

```
slti rt, rs, immediate
```

(27) **sltiu**

无符号数比较，如果 rs < immediate，则 rt = 1，否则 rt = 0
（rs零扩展1位，立即数符号扩展为32位后，再零扩展1位）

```
sltiu rt, rs, immediate
```

(28) **lui**

立即数加载至高位：将 immediate 左移 16 位，结果存入 rt 中

```
lui rt, immediate
```

（三）加载

加载是从**内存**中读取数据到寄存器中，指令的格式如下：

```
OP rt, offset(rs)
```

其中，OP为操作码，**rt为目标寄存器，rs为基地址，offset是一个16位的有符号立即数，计算时其进行符号扩展，表示相对于基地址的偏移量。**

(29) **lb**

有符号加载字节：内存地址为 rs + offset ，从内存中读取8位数据到寄存器rt中，并进行符号扩展

```
lb rt,  offset(rs)
```

(30) **lbu**

无符号加载字节，内存地址为 rs + offset ，从内存中读取8位数据到寄存器rt中，并进行零扩展

```
lbu rt,  offset(rs)
```

(31) **lh**

有符号加载半字，内存地址为 rs + offset ，从内存中读取16位数据到寄存器rt中，并进行符号扩展
（内存的地址和2对齐，即最低位为0）

```
lh rt,  offset(rs)
```

(32) **lhu**

无符号加载半字，内存地址为 rs + offset ，从内存中读取16位数据到寄存器rt中，并进行零扩展
（内存的地址和2对齐，即最低位为0）

```
lhu rt,  offset(rs)
```

(33) **lw**

加载字，内存地址为 rs + offset ，从内存中读取32位数据到寄存器rt中
（内存的地址和4对齐，即最低两位为00）

```
lw rt,  offset(rs)
```

（四）存储

存储是将寄存器中的数据写入内存中，其指令的格式如下：

```
OP rt, offset(rs)
```

其中，OP为操作码，**rt为源寄存器**，**rs为基地址，offset是一个16位的有符号立即数，表示相对于基地址的偏移量。**

(34) **sb**

存储字节，将 rt 寄存器的最低字节存入地址为 rs + offset 的内存中

```
sb rt,  offset(rs)
```

(35) **sh**

存储半字，将 rt 寄存器的最低半字存入地址为 rs + offset 的内存中
（内存的地址和2对齐，即最低位为0）

```
sh rt,  offset(rs)
```

(36) **sw**

存储字，将 rt 寄存器的最低字存入地址为 rs + offset 的内存中
（内存的地址和4对齐，即最低两位为00）

```
sw rt,  offset(rs)
```

（五）分支

分支指令用于实现程序流程的控制，其指令的格式如下：

```
OP rs, rt, offset
```

其中，OP为操作码，**rs和rt为源寄存器**，**offset是一个16位的有符号立即数，表示相对于PC的偏移量。**（PC是程序计数器，存储指向下一条指令的地址）

(37) **beq**

相等时跳转：**如果 rs = rt，pc = pc + 4 + { offset , 00 }，否则pc=pc+4**
（ offset 是指令的偏移量，因为操作过程中，一条指令占 4 字节，用地址访存的话需要跳转 4*offet 字节，所以offet需要拼接两位 0 ）

```
beq rs, rt, offset
```

(38) **bne**

不相等时跳转：如果 rs != rt，pc = pc + 4 + { offset , 00 }，否则pc=pc+4

```
bne rs, rt, offset
```

(39) **blez**

小于等于零时跳转：如果 rs <= 0，pc = pc + 4 + { offset , 00 }，否则pc=pc+4

```
blez rs, offset
```

(40) **bgtz**

大于零时跳转：如果 rs > 0，pc = pc + 4 + { offset , 00 }，否则pc=pc+4

```
bgtz rs, offset
```

(41) **bltz**

小于零时跳转：如果 rs < 0，pc = pc + 4 + { offset , 00 }，否则pc=pc+4

```
bltz rs, offset
```

(42) **bgez**

大于等于零时跳转：如果 rs >= 0，pc = pc + 4 + { offset , 00 }，否则pc=pc+4

```
bgez rs, offset
```

（六）跳转

跳转指令用于实现无条件跳转，其指令的格式如下：

```
OP target
```

其中，OP为操作码，**target是一个标签，表示跳转的目标地址。**

(43) **j**

跳转：j 指令是 PC 相关的转移指令。当把 4GB 划分为 16 个 256MB 区域，j 指令可以在当前PC 所在的 256MB 区域内任意跳转。（如果需要跳转范围超出了当前 PC 所在的 256MB 区域内时，可以使用 JR 指令）

```
j target
```

(44) **jal**

跳转并链接：jal 指令是函数指令，PC 转向被调用函数，同时将当前 PC+4 保存在 GPR[31]中。当把4GB 划分为 16 个 256MB 区域，jal 指令可以在当前 PC 所在的 256MB 区域内任意跳转。 (jal 与 jr 配套使用。jal 用于调用函数，jr 用于函数返回。当所调用的函数地址超出了当前 PC 所在的 256MB 区域内时，可以使用 jalr 指令。)

```
jal target
```

(45) **jr**

跳转至寄存器：jr 指令是 PC 相关的转移指令。将 GPR[rs] 的值赋给 PC，实现跳转。当把 4GB 划分为 16 个 256MB 区域，jr 指令可以在当前 PC 所在的 256MB 区域内任意跳转。（jr 与 jal 配套使用。jr 用于函数返回，jal 用于调用函数。当所调用的函数地址超出了当前 PC 所在的 256MB 区域内时，可以使用 jalr 指令。）

```
jr rs
```

(46) **jalr**

跳转并链接：jalr 指令是函数指令，PC 转向被调用函数(函数入口地址保存在 GPR[rs]中)，同时将当前 PC+4 保存在 GPR[rd]中。

```
jalr rd, rs
```

（七）传输

传输指令用于实现寄存器和内存之间的数据传输，其指令的格式如下：

```
OP rt, rs
```

其中，OP为操作码，**rt为目标寄存器，rs为源寄存器。**

(47) **mfhi**

读HI寄存器：将 HI 寄存器的值传送到 rt 寄存器中。
（HI 寄存器用于存储乘法/除法运算的中间结果的高 32 位，当乘法/除法计算完毕后，需要用 mfhi 读取相应的结果。）

```
mfhi rt
```

(48) **mflo**

读LO寄存器：将 LO 寄存器的值传送到 rt 寄存器中。
（LO 寄存器用于存储乘法/除法运算的中间结果的低 32 位，当乘法/除法计算完毕后，需要用 mflo 读取相应的结果。）

```
mflo rt
```

(49) **mthi**

写HI寄存器：将 rt 寄存器的值传送到 HI 寄存器中。
（HI 寄存器用于存储乘法/除法运算的中间结果的高 32 位，当乘法/除法计算完毕后，需要用 mfhi 读取相应的结果。）

```
mthi rs
```

(50) **mtlo**

写LO寄存器：将 rt 寄存器的值传送到 LO 寄存器中。
（LO 寄存器用于存储乘法/除法运算的中间结果的低 32 位，当乘法/除法计算完毕后，需要用 mflo 读取相应的结果。）

```
mtlo rs
```

（八）特权

特权指令用于系统调用

(51) **eret**

异常返回：eret 指令用于从异常处理程序返回到异常发生前的程序。当异常处理程序执行完毕后，eret 指令会将 PC 寄存器的值设置为 EPC 寄存器的值（被中断指令的下一条地址），从而实现返回。

```
eret
```

(52) **mfc0**

读CP0寄存器：将CP0的寄存器 ct 的值传送到 rt 寄存器中。
（CP0用于实现系统调用，当系统调用结束后，需要用 mfc0 读取相应的结果。）

```
mfc0 rt, ct
```

(53) **mtc0**

写CP0寄存器：将 rt 寄存器的值传送到 CP0的寄存器 ct 中。
（CP0用于实现系统调用，当系统调用结束后，需要用 mfc0 读取相应的结果。）

```
mtc0 rt, ct
```

（九）陷阱

陷阱指令用于系统调用和调试

(54) **syscall**

系统调用：syscall 指令用于实现系统调用，当程序执行到 syscall 指令时，会根据当前的 $v0 寄存器的值（称为系统调用号），进行相应的操作，后文将会详细论述

```
syscall
```

(55) **break**

断点指令：break 指令用于调试程序，当程序执行到 break 指令时，会触发一个异常，进入异常处理程序。异常处理程序会记录程序的状态，并暂停程序的执行。

```
break
```

（十） 扩展指令

(56) **li**

加载立即数：将立即数加载到寄存器中。
如果立即数不超过16位，那么汇编器会将其转换为一条addi指令。如果超过了16位数值太大，无法直接作为立即数操作，那么汇编器会使用lui（加载上半字）和ori（或立即数）两条指令来组合成一个32位的数值

```
li rt, immediate
```

(57) **la**

加载地址：将标签的地址加载到寄存器中，标签的地址是32位的有符号数。

```
la rt, label
```

(58) **move**

移动：将寄存器rs的值赋给另一个寄存器rt。

```
move rt, rs
```

二、伪指令

(1) .data

定义程序的数据段，初始地址为 address，若无 address 参数，初始地址为设置的默认地址。 需要用伪指令声明的程序变量需要紧跟着该指令。

```
.data [address]
```

(2) .text

定义程序的代码段，初始地址为address，若无 address 参数，初始地址为设置的默认地址。 该指令后面就是程序代码。
在 MARS 中如果前面没有使用 .data 伪指令，可以不使用 .text 直接编写程序代码，代码将放置在前面设置的代码段默认地址中，**但如果前面使用了 .data 伪指令，务必在代码段开始前使用 .text 进行标注。**

```
.text [address]
```

(3) .space

申请 n 个字节未初始化的内存空间，类似于其他语言中的数组声明。这段数据的初始地址保存在标签 name 中。
name 的地址是由 .data 段的初始地址加上前面所申请的数据大小计算得出的。由于前面申请的空间大小不定，有可能会出现后来申请的空间没有字对齐的情况，从而在使用 sw,lw 一类指令时出现错误，所以在申请空间时尽可能让 n 为 4 的倍数，防止在数据存取过程中出现问题。

```
name: .space n
```

(4) .word

在内存数据段中以字为单位连续存储数据 data1, data2,... (也就是将 datax 写入对应的 1 个字的空间，注意 .word 和 .space 的区别: .word 是将数据写入空间，而 .space 是申请空间但不写入数据)
这段数据的初始地址保存在标签 name 中。计算方式与上面相同。

```
.word data1, data2, ...
```

(5) .byte

以字节为单位存储数据 data1, data2,... (也就是将 datax 写入对应的 1 个字节的空间，注意 .byte 和 .word 的区别: .byte 是将数据写入1个字节空间，而 .word 是将数据写入 4 个字节的空间)

这段数据的初始地址保存在标签 `name` 中。计算方式与上面相同。
由于是按字节写入，可能会导致之后分配的空间首地址无法字对齐的情况发生，此时需要使用 `.align` 指令进行对齐

```
.byte data1, data2, ...
```

(6) .align

对齐指令，将数据段对齐到 2 的幂次方大小，即 2^N 字节，`N` 为参数。

```
.align N
```

(7) .asciiz

以字节为单位存储字符串`String`，末尾以 `NULL` 结尾。
这个字符串在内存数据区的初始地址保存在标签 `name` 中。
`.asciiz` 由于是按字节存储，可能会导致之后分配的空间首地址无法字对齐的情况发生，此时需要使用 `.align` 指令进行对齐。

```
name: .asciiz "string"
.align 2
```

(8) .ascii

与 `.asciiz` 类似，但字符串末尾不添加 `NULL` 结尾。因此如果连续多次使用 `.ascii`，那么这几个字符串将会拼接在一起

```
name: .ascii "string"
```

三、指令跳转范围

- **1个地址单元存储有有8 bits（8比特）**
- **8 bits = 1 B（1字节）**
- **1024 B = 1 KB（1千字节）**
- **1024 KB = 1 MB（1兆字节）**
- **1024 MB = 1 GB（1吉字节）**
- **1024 GB = 1 TB（1太字节）**

- **2^10 = 1024**

(1) j 指令

j 指令 有26位用于存储跳转的地址，当跳转时，低位补两位0（因为指令存储地址时**字（32bits）对齐**的），高位补PC+4的高4位（伪直接寻址），故跳转范围由26位决定。 故，可跳转2^(26+2)=2^28个地址单元，即256 MB。

(2) jr 指令

jr 指令 采用寄存器寻址方式寻址，跳转到$ra寄存器保存的地址，因为$ra寄存器保存32位数据，故可跳转地址范围大小是2^32个地址单元，即4GB。

(3) beq 等分支指令

beq指令跳转到 label 所指的代码，采用PC相对寻址。beq指令中立即数保存的是（label 所指代码行行数）-（当前代码行+1），寻址跳转时对imm左移两位（相当于乘4，由行数差转变为地址单元差）再进行符号扩展，最后加（PC + 4）得到最终要跳转到的地址单元。故跳转代码范围是2^(16+2)个地址单元，即 256 KB。

四、*syscall* 指令详解

> 在执行 syscall 指令时，MIPS 处理器会将控制权交给操作系统内核，操作系统内核会根据寄存器 `$v0` 中的值来确定要执行的系统调用的类型。`$v0` **寄存器在执行 syscall 之前应该被设置为系统调用的编号**。系统调用完成后，操作系统会将返回值存放在 `$v0` 寄存器中。
> 以下是一些常见的 MIPS 系统调用号及其对应的操作： 在 MIPS 系统中，系统调用号用于指定执行哪种系统调用。不同的系统调用号对应不同的服务和操作。以下是一些常见的 MIPS 系统调用号及其对应的操作：

1. **打印整数（print_int）** - 系统调用号 1
    - 将整数输出到控制台。
    - 寄存器 `$a0` 包含要打印的整数值。
2. **打印浮点数（print_float）** - 系统调用号 2
    - 将浮点数输出到控制台。
    - 寄存器 `$f12` 包含要打印的浮点数值。
3. **打印双精度数（print_double）** - 系统调用号 3
    - 将双精度数输出到控制台。
    - 寄存器 `$f12` 包含要打印的双精度数值。
4. **打印字符串（print_string）** - 系统调用号 4
    - 将字符串输出到控制台。

- 寄存器 `$a0` 包含字符串的地址。

5. **读取整数（read_int）** - 系统调用号 5
   - 从标准输入读取一个整数。
   - 读取的整数值被存储在寄存器 `$v0` 中。
6. **读取浮点数（read_float）** - 系统调用号 6
   - 从标准输入读取一个浮点数。
   - 读取的浮点数值被存储在寄存器 `$f0` 中。
7. **读取双精度数（read_double）** - 系统调用号 7
   - 从标准输入读取一个双精度数。
   - 读取的双精度数值被存储在寄存器 `$f0` 中。
8. **读取字符串（read_string）** - 系统调用号 8
   - 从标准输入读取一个字符串。
   - 寄存器 `$a0` 包含缓冲区的地址，`$a1` 包含要读取的字符数。
9. **内存分配（sbrk）** - 系统调用号 9
   - 改变进程的可用内存量。
   - 寄存器 `$a0` 包含请求的字节数，返回值是新分配内存的地址。
10. **退出（exit）** - 系统调用号 10
    - 终止程序的执行。
    - 寄存器 `$a0` 可以包含退出状态码。
11. **打印字符（print_character）** - 系统调用号 11
    - 打印单个字符到控制台。
    - 寄存器 `$a0` 包含要打印的字符。
12. **读取字符（read_character）** - 系统调用号 12
    - 从标准输入读取一个字符。
    - 读取的字符被存储在寄存器 `$v0` 中。