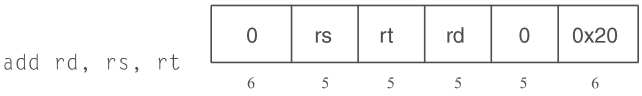
campos adicionais para distinguir instruções relacionadas. Por exemplo, as diferentes instruções de ponto flutuante são especificadas pelos bits 0-5. As setas a partir da primeira coluna mostram quais opcodes utilizam esses campos adicionais.

Formato de instrução

O restante deste apêndice descreve as instruções implementadas pelo hardware MIPS real e as pseudo-instruções fornecidas pelo montador MIPS. Os dois tipos de instruções podem ser distinguidos facilmente. As instruções reais indicam os campos em sua representação binária. Por exemplo, em

Adição (com overflow)



a instrução `add` consiste em seis campos. O tamanho de cada campo em bits é o pequeno número abaixo do campo. Essa instrução começa com 6 bits em 0. Os especificadores de registradores começam com um *r*, de modo que o próximo campo é um especificador de registrador de 5 bits chamado `rs`. Esse é o mesmo registrador que é o segundo argumento no assembly simbólico à esquerda dessa linha. Outro campo comum é $\texttt{imm}_{16}$, que é um número imediato de 16 bits.

As pseudo-instruções seguem aproximadamente as mesmas convenções, mas omitem a informação de codificação de instrução. Por exemplo:

Multiplicação (sem overflow)

```
mul rdest, rsrcl, src2        pseudo-instrução
```

Nas pseudo-instruções, `rdest` e `rsrc1` são registradores, e `src2` é um registrador ou um valor imediato. Em geral, o montador e o SPIM traduzem uma forma mais geral de uma instrução (por exemplo, `add $v1, $a0, 0x55`) para uma forma especializada (por exemplo, `addi $v1, $a0, 0x55`).

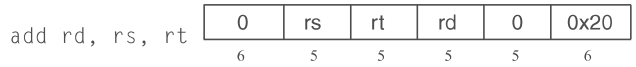
Instruções aritméticas e lógicas

Valor absoluto

Coloca o valor absoluto do registrador `rsrc` no registrador `rdest`.

```
abs, rdest, rsrc        pseudo-instrução
```

Adição (com overflow)

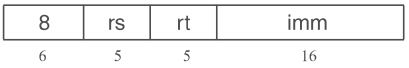


Adição (sem overflow)

Coloca a soma dos registradores `rs` e `rt` no registrador `rd`.

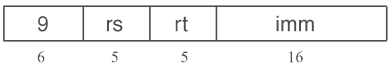
Adição imediato (com overflow)

```
addi rt, rs, imm
```



Adição imediato (sem overflow)

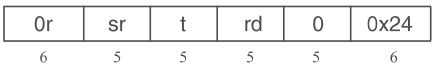
```
addiu rt, rs, imm
```



Coloca a soma do registrador rs e o imediato com sinal estendido no registrador `rt`.

AND

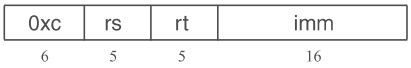
```
and rd, rs, rt
```



Coloca o AND lógico dos registradores `rs` e `rt` no registrador `rd`.

AND imediato

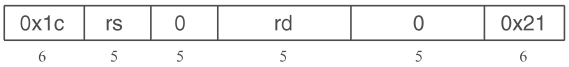
```
andi rt, rs, imm
```



Coloca o AND lógico do registrador `rs` e o imediato estendido com zeros no registrador `rt`.

Contar uns iniciais

```
clo rd, rs
```



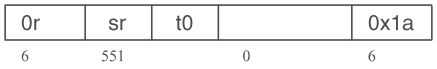
Contar zeros iniciais

```
clz rd, rs
```

Conta o número de uns (zeros) iniciais da word no registrador `rs` e coloca o resultado no registrador `rd`. Se uma word contém apenas uns (zeros), o resultado é 32.

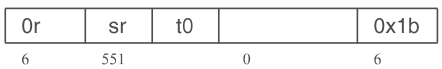
Divisão (com overflow)

```
div rs, rt
```



Divisão (sem overflow)

```
divu rs, rt
```



Divide o registrador `rs` pelo registrador `rt`. Deixa o quociente no registrador `lo` e o resto no registrador `hi`. Observe que, se um operando for negativo, o restante não será especificado pela arquitetura MIPS e dependerá da convenção da máquina em que o SPIM é executado.

Divisão (com overflow)

`div rdest, rsrc1, src2` *pseudo-instrução*

Divisão (sem overflow)

`divu rdest, rsrc1, src2` *pseudo-instrução*

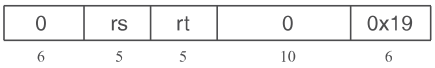
Coloca o quociente do registrador `rsrc1` pelo `src2` no registrador `rdest`.

Multiplicação

`mult rs, rt`

Multiplicação sem sinal

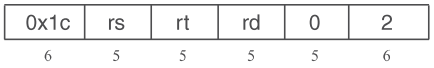
`multu rs, rt`



Multiplica os registradores `rs` e `rt`. Deixa a word menos significativa do produto no registrador `lo` e a word mais significativa no registrador `hi`.

Multiplicação (sem overflow)

`mul rd, rs, rt`



Coloca os 32 bits menos significativos do produto de `rs` e `rt` no registrador `rd`.

Multiplicação (com overflow)

`mulo rdest, rsrc1, src2` *pseudo-instrução*

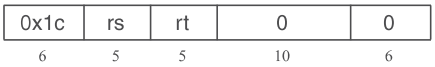
Multiplicação sem sinal (com overflow)

`mulou rdest, rsrc1, src2` *pseudo-instrução*

Coloca os 32 bits menos significativos do produto do registrador `rsrc1` e `src2` no registrador `rdest`.

Multiplicação adição

`madd rs, rt`



Multiplicação adição sem sinal

`maddu rs, rt`

Multiplica os registradores `rs` e `rt` e soma o produto de 64 bits resultante ao valor de 64 bits nos registradores concatenados `lo` e `hi`.

Multiplicação subtração

```
msub rs, rt
```

Multiplicação subtração sem sinal

```
msub rs, rt
```

Multiplica os registradores `rs` e `rt` e subtrai o produto de 64 bits resultante do valor de 64 bits nos registradores concatenados `lo` e `hi`.

Negar valor (com overflow)

```
neg rdest, rsrc
```
pseudo-instrução

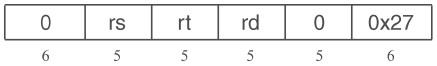
Negar valor (sem overflow)

```
negu rdest,rsrc
```
pseudo-instrução

Coloca o negativo do registrador `rsrc` no registrador `rdest`.

NOR

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



Coloca o NOR lógico dos registradores `rs` e `rt` para o registrador `rd`.

NOT

```
not rdest, rsrc
```
pseudo-instrução

Coloca a negação lógica bit a bit do registrador `rsrc` no registrador `rdest`.

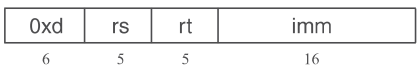
OR

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

Coloca o OR lógico dos registradores `rs` e `rt` no registrador `rd`.

OR imediato

```
ori rt, rs, imm
```



Coloca o OR lógico do registrador `rs` e o imediato estendido com zero no registrador `rt`.

Resto

`rem rdest, rsrc1, rsrc2` *pseudo-instrução*

Resto sem sinal

`remu rdest, rsrc1, rsrc2` *pseudo-instrução*

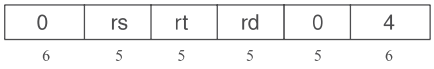
Coloca o resto do registrador `rsrc1` dividido pelo registrador `rsrc2` no registrador `rdest`. Observe que se um operando for negativo, o resto não é especificado pela arquitetura MIPS e depende da convenção da máquina em que o SPIM é executado.

Shift lógico à esquerda

`sll rd, rt, shamt`

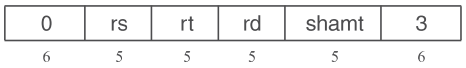
Shift lógico à esquerda variável

`sllv rd, rt, rs`



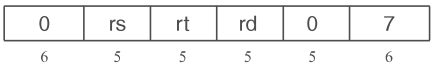
Shift aritmético à direita

`sra rd, rt, shamt`



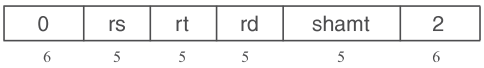
Shift aritmético à direita variável

`srav rd, rt, rs`



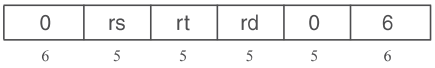
Shift lógico à direita

`srl rd, rt, shamt`



Shift lógico à direita variável

`srlv rd, rt, rs`



Desloca o registrador `rt` à esquerda (direita) pela distância indicada pelo `shamt` imediato ou pelo registrador `rs` e coloca o resultado no registrador `rd`. Observe que o argumento `rs` é ignorado para `sll`, `sra` e `srl`.

Rotate à esquerda

`rol rdest, rsrc1, rsrc2` *pseudo-instrução*

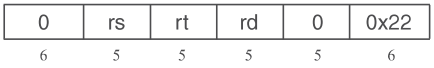
Rotate à direita

`ror rdest, rsrc1, rsrc2` *pseudo-instrução*

Gira o registrador `rsrc1` à esquerda (direita) pela distância indicada por `rsrc2` e coloca o resultado no registrador `rdest`.

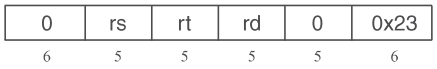
Subtração (com overflow)

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



Subtração (sem overflow)

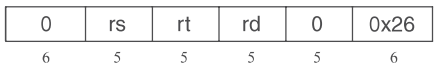
```
subu rd, rs, rt
```



Coloca a diferença dos registradores `rs` e `rt` no registrador `rd`.

OR exclusivo

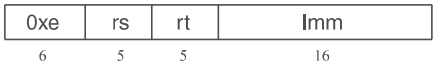
```
xor rd, rs, rt
```



Coloca o XOR lógico dos registradores `rs` e `rt` no registrador `rd`.

XOR imediato

```
xori rt, rs, imm
```

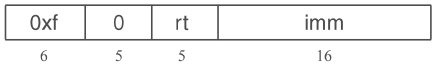


Coloca o XOR lógico do registrador `rs` e o imediato estendido com zeros no registrador `rt`.

Instruções para manipulação de constantes

Load superior imediato

```
lui rt, imm
```



Carrega a halfword menos significativa do imediato `imm` na halfword mais significativa do registrador `rt`. Os bits menos significativos do registrador são colocados em 0.

Load imediato

```
li rdest, imm
```

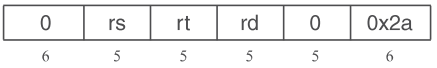
pseudo-instrução

Move o imediato `imm` para o registrador `rdest`.

Instruções de comparação

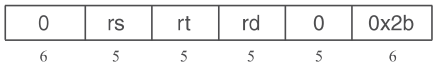
Set se menor que

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



Set se menor que sem sinal

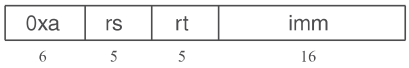
```
sltu rd, rs, rt
```



Coloca o registrador `rd` em 1 se o registrador `rs` for menor que `rt`; caso contrário, coloca-o em 0.

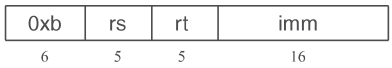
Set se menor que imediato

```
slti rt, rs, imm
```



Set se menor que imediato sem sinal

```
sltiu rt, rs, imm
```



Coloca o registrador `rt` em 1 se o registrador `rs` for menor que o imediato estendido com sinal, e em 0 em caso contrário.

Set se igual

```
seq rdest, rsrc1, rsrc2
```
pseudo-instrução

Coloca o registrador `rdest` em 1 se o registrador `rsrc1` for igual a `rsrc2`, e em 0 caso contrário.

Set se maior ou igual

```
sge rdest, rsrc1, rsrc2
```
pseudo-instrução

Set se maior ou igual sem sinal

```
sgeu rdest, rsrc1, rsrc2
```
pseudo-instrução

Coloca o registrador `rdest` em 1 se o registrador `rsrc1` for maior ou igual a `rsrc2`, e em 0 caso contrário.

Set se maior que

```
sgt rdest, rsrc1, rsrc2
```
pseudo-instrução

Set se maior que sem sinal

```
sgtu rdest, rsrc1, rsrc2
```
pseudo-instrução

Coloca o registrador `rdest` em 1 se o registrador `rsrc1` for maior que `rsrc2`, e em 0 caso contrário.

Set se menor ou igual

```
sle rdest, rsrc1, rsrc2
```
pseudo-instrução

Set se menor ou igual sem sinal

```
sleu rdest, rsrc1, rsrc2
```
pseudo-instrução

Coloca o registrador `rdest` em 1 se o registrador `rsrc1` for menor ou igual a `rsrc2`, e em 0 caso contrário.

Set se diferente

```
sne rdest, rsrc1, rsrc2
```
pseudo-instrução

Coloca o registrador `rdest` em 1 se o registrador `rsrc1` não for igual a `rsrc2`, e em 0 caso contrário.

Instruções de desvio

As instruções de desvio utilizam um campo *offset* de instrução de 16 bits com sinal; logo, elas podem desviar $2^{15} - 1$ *instruções* (não bytes) para a frente ou 2^{15} instruções para trás. A instrução *jump* contém um campo de endereço de 26 bits. Em processadores MIPS reais, as instruções de desvio são delayed branches, que não transferem o controle até que a instrução após o desvio (seu "delay slot") tenha sido executado (ver Capítulo 6). Os delayed branches afetam o cálculo de offset, pois precisam ser calculados em relação ao endereço da instrução do delay slot (PC + 4), que é quando o desvio ocorre. O SPIM não simula esse delay slot, a menos que os flags `-bare` ou `-delayed_branch` sejam especificados.

No código assembly, os offsets normalmente não são especificados como números. Em vez disso, uma instrução desvia para um rótulo, e o montador calcula a distância entre o desvio e a instrução destino.

No MIPS32, todas as instruções de desvio condicional reais (não pseudo) têm uma variante "provável" (por exemplo, a variável provável de `beq` é `beql`), que *não* executa a instrução no delay slot do desvio se o desvio não for tomado. Não use essas instruções; elas poderão ser removidas em versões subseqüentes da arquitetura. O SPIM implementa essas instruções, mas elas não são descritas daqui por diante.

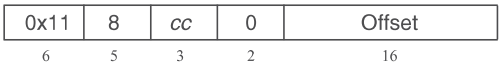
Branch

```
b label
```
pseudo-instrução

Desvia incondicionalmente para a instrução no rótulo.

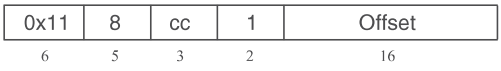
Branch co-processador falso

```
bc1f cc label
```



Branch co-processador verdadeiro

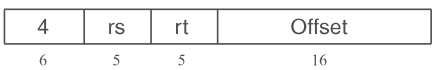
```
bc1t cc label
```



Desvia condicionalmente pelo número de instruções especificado pelo offset se o flag de condição de ponto flutuante numerado como *cc* for falso (verdadeiro). Se *cc* for omitido da instrução, o flag de código de condição 0 é assumido.

Branch se for igual

```
beq rs, rt, label
```



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` for igual a `rt`.

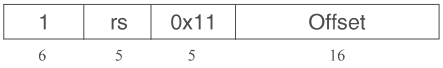
Branch se for maior ou igual a zero

```
bgez rs, label
```

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` for maior ou igual a 0.

Branch se for maior ou igual a zero e link

```
bgezal rs, label
```



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` for maior ou igual a 0. Salva o endereço da próxima instrução no registrador 31.

Branch se for maior que zero

```
bgtz rs, label
```

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` for maior que 0.

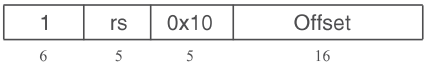
Branch se for menor ou igual a zero

```
blez rs, label
```

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` for menor ou igual a 0.

Branch se for menor e link

```
bltzal rs, label
```



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` for menor que 0. Salva o endereço da próxima instrução no registrador 31.

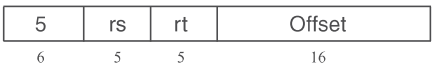
Branch se for menor que zero

```
bltz rs, label
```

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` for menor que 0.

Branch se for diferente

`bne rs, rt, label`



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` não for igual a `rt`.

Branch se for igual a zero

`beqz rsrc, label` *pseudo-instrução*

Desvia condicionalmente para a instrução no rótulo se `rsrc` for igual a 0.

Branch se for maior ou igual

`bge rsrc1, rsrc2, label` *pseudo-instrução*

Branch se for maior ou igual com sinal

`bgeu rsrc1, rsrc2, label` *pseudo-instrução*

Desvia condicionalmente até a instrução no rótulo se o registrador `rsrc1` for maior ou igual a `rsrc2`.

Branch se for maior

`bgt rsrc1, src2, label` *pseudo-instrução*

Branch se for maior sem sinal

`bgtu rsrc1, src2, label` *pseudo-instrução*

Desvia condicionalmente para a instrução no rótulo se o registrador `rsrc1` for maior do que `src2`.

Branch se for menor ou igual

`ble rsrc1, src2, label` *pseudo-instrução*

Branch se for menor ou igual sem sinal

`bleu rsrc1, src2, label` *pseudo-instrução*

Desvia condicionalmente para a instrução no rótulo se o registrador `rsrc1` for menor ou igual a `rsrc2`.

Branch se for menor

`blt rsrc1, rsrc2, label` *pseudo-instrução*

Branch se for menor sem sinal

`bltu rsrc1, rsrc2, label` *pseudo-instrução*

Desvia condicionalmente para a instrução no rótulo se o registrador `rsrc1` for menor do que `src2`.

Branch se não for igual a zero

```
bnez rsrc, label
```
pseudo-instrução

Desvia condicionalmente para a instrução no rótulo se o registrador `rsrc` não for igual a 0.

Instruções de jump

Jump

```
j target
```

Desvia incondicionalmente para a instrução no destino.

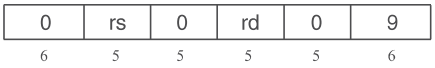
Jump-and-link

```
jal target
```

Desvia incondicionalmente para a instrução no destino. Salva o endereço da próxima instrução no registrador `$ra`.

Jump-and-link registrador

```
jalr rs, rd
```



Desvia incondicionalmente para a instrução cujo endereço está no registrador `rs`. Salva o endereço da próxima instrução no registrador `rd` (cujo default é 31).

Jump registrador

```
jr rs
```

Desvia incondicionalmente para a instrução cujo endereço está no registrador `rs`.

Instruções de trap

Trap se for igual

```
teq rs, rt
```

Se o registrador `rs` for igual ao registrador `rt`, gera uma exceção de Trap.

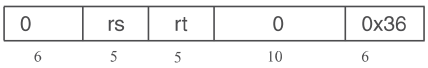
Trap se for igual imediato

```
teqi rs, imm
```

Se o registrador `rs` for igual ao valor de `imm` com sinal estendido, gera uma exceção de Trap.

Trap se não for igual

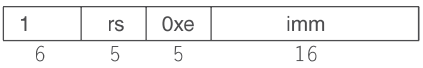
```
teq rs, rt
```



Se o registrador `rs` não for igual ao registrador `rt`, gera uma exceção de Trap.

Trap se não for igual imediato

```
teqi rs, imm
```



Se o registrador `rs` não for igual ao valor de `imm` com sinal estendido, gera uma exceção de Trap.

Trap se for maior ou igual

```
tge rs, rt
```

Trap sem sinal se for maior ou igual

```
tgeu rs, rt
```

Se o registrador `rs` for maior ou igual ao registrador `rt`, gera uma exceção de Trap.

Trap se for maior ou igual imediato

```
tgei rs, imm
```

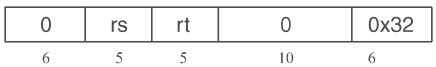
Trap sem sinal se for maior ou igual imediato

```
tgeiu rs, imm
```

Se o registrador `rs` for maior ou igual ao valor de `imm` com sinal estendido, gera uma exceção de Trap.

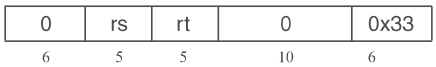
Trap se for menor

```
tlt rs, rt
```



Trap sem sinal se for menor

```
tltu rs, rt
```



Se o registrador `rs` for menor que o registrador `rt`, gera uma exceção de Trap.

Trap se for menor imediato

```
tlti rs, imm
```

Trap sem sinal se for menor imediato

`tltiu rs, imm`

Se o registrador `rs` for menor do que o valor de `imm` com sinal estendido, gera uma exceção de Trap.

Instruções load

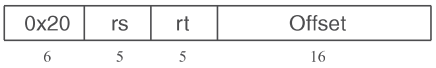
Load endereço

`la rdest, address` *pseudo-instrução*

Carrega o *endereço* calculado – não o conteúdo do local – para o registrador `rdest`.

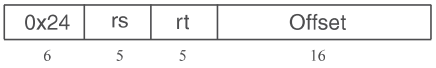
Load byte

`lb rt, address`



Load byte sem sinal

`lbu rt, address`



Carrega o byte no *endereço* para o registrador `rt`. O byte tem sinal estendido por `lb`, mas não por `lbu`.

Load halfword

`lh rt, address`

Load halfword sem sinal

`lhu rt, address`

Carrega a quantidade de 16 bits (halfword) no *endereço* para o registrador `rt`. A halfword tem sinal estendido por `lh`, mas não por `lhu`.

Load word

`lw rt, address`

Carrega a quantidade de 32 bits (word) no *endereço* para o registrador `rt`.

Load word co-processador 1

`lwc1 ft, address`

Carrega a word no *endereço* para o registrador `ft` da unidade de ponto flutuante.

Load word à esquerda

```
lwl rt, address
```

Load word à direita

```
lwr rt, address
```

Carrega os bytes da esquerda (direita) da word do *endereço* possivelmente não alinhado para o registrador `rt`.

Load doubleword

```
ld rdest, address
```
pseudo-instrução

Carrega a quantidade de 64 bits no *endereço* para os registradores `rdest` e `rdest + 1`.

Load halfword não alinhada

```
ulh rdest, address
```
pseudo-instrução

Load halfword sem sinal não alinhada

```
ulhu rdest, address
```
pseudo-instrução

Carrega a quantidade de 16 bits (halfword) no *endereço* possivelmente não alinhado para o registrador `rdest`. A halfword tem extensão de sinal por `ulh`, mas não `ulhu`.

Load word não alinhada

```
ulw rdest, address
```
pseudo-instrução

Carrega a quantidade de 32 bits (word) no *endereço* possivelmente não alinhado para o registrador `rdest`.

Load Linked

```
ll rt, address
```

Carrega a quantidade de 32 bits (word) no *endereço* para o registrador `rt` e inicia uma operação ler-modificar-escrever indivisível. Essa operação é concluída por uma instrução de armazenamento condicional (`sc`), que falhará se outro processador escrever no bloco que contém a word carregada. Como o SPIM não simula processadores múltiplos, a operação de armazenamento condicional sempre tem sucesso.

Instruções store

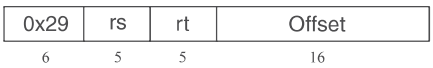
Store byte

```
sb rt, address
```

Armazena o byte baixo do registrador `rt` no *endereço*.

Store halfword

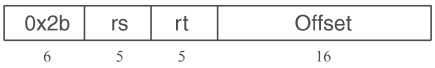
```
sh rt, address
```



Armazena a halfword baixa do registrador `rt` no *endereço*.

Store word

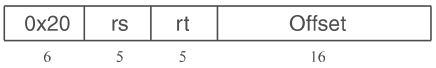
```
sw rt, address
```



Armazena a word do registrador `rt` no *endereço*.

Store word co-processador 1

```
lb rt, address
```



Armazena o valor de ponto flutuante no registrador `ft` do co-processador de ponto flutuante no *endereço*.

Store double co-processador 1

```
sdc1 ft, address
```

Armazena o valor de ponto flutuante da dupla word nos registradores `ft` e `ft + 1` do co-processador de ponto flutuante em *endereço*. O registrador `ft` precisa ser um número par.

```
swl rt, address
```

Store word à esquerda

```
swr rt, address
```

Store word à direita

Armazena os bytes da esquerda (direita) do registrador `rt` no *endereço* possivelmente não alinhado.

Store doubleword

```
sd rsrc, address
```
pseudo-instrução

Armazena a quantidade de 64 bits nos registradores `rsrc` e `rsrc + 1` no *endereço*.

Store halfword não alinhada

```
ush rsrc, address
```
pseudo-instrução

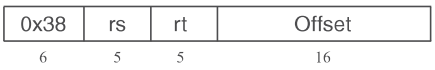
Armazena a halfword baixa do registrador `rsrc` no *endereço* possivelmente não alinhado.

Store word não alinhada

```
usw rsrc, address
```
pseudo-instrução

Armazena a word do registrador `rsrc` no *endereço* possivelmente não alinhado.

`sc rt, address`



Store condicional

Armazena a quantidade de 32 bits (word) no endereço `rt` para a memória no *endereço* e completa uma operação ler-modificar-escrever indivisível. Se essa operação indivisível tiver sucesso, a word da memória será modificada e o registrador `rt` será colocado em 1. Se a operação indivisível falhar porque outro processador escreveu em um local no bloco contendo a word endereçada, essa instrução não modifica a memória e escreve 0 no registrador `rt`. Como o SPIM não simula diversos processadores, a instrução sempre tem sucesso.

Instruções para movimentação de dados

Move

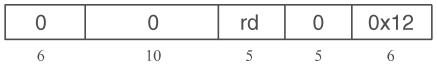
`move rdest, rsrc` *pseudo-instrução*

Move o registrador `rsrc` para `rdest`.

`mfhi rd`

Move de hi

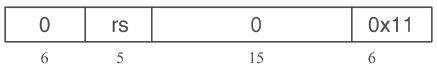
`mflo rd`



Move de lo

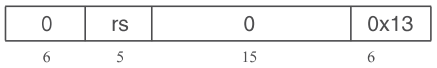
A unidade de multiplicação e divisão produz seu resultado em dois registradores adicionais, `hi` e `lo`. Essas instruções movem os valores de e para esses registradores. As pseudo-instruções de multiplicação, divisão e resto que fazem com que essa unidade pareça operar sobre os registradores gerais movem o resultado depois que o cálculo terminar. Move o registrador `hi` (`lo`) para o registrador `rd`.

`mthi rs`



Move para hi

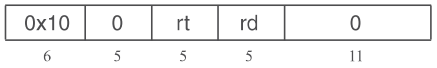
`mtlo rs`



Move para lo

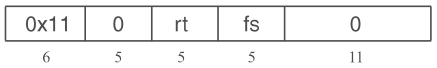
Move o registrador `rs` para o registrador `hi` (`lo`).

`mfc0 rt, rd`



Move do co-processador 0

`mfc1 rt, fs`



Move do co-processador 1

Os co-processadores têm seus próprios conjuntos de registradores. Essas instruções movem valores entre esses registradores e os registradores da CPU. Move o registrador `rd` em um co-processador (registrador `fs` na FPU) para o registrador `rt` da CPU. A unidade de ponto flutuante é o co-processador 1.

Move double do co-processador 1

```
mfc1.d rdest, frsrc1        pseudo-instrução
```

Move os registradores de ponto flutuante `frsrc1` e `frsrc1 + 1` para os registradores da CPU `rdest` e `rdest + 1`.

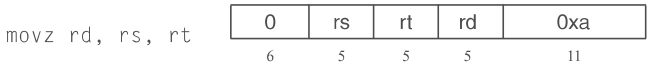
Move para co-processador 0

Move para co-processador 1

Move o registrador da CPU `rt` para o registrador `rd` em um co-processador (registrador `fs` na FPU).

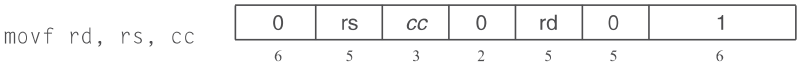
Move condicional diferente de zero

Move o registrador `rs` para o registrador `rd` se o registrador `rt` não for 0.



Move condicional zero

Move o registrador `rs` para o registrador `rd` se o registrador `rt` for 0.



Move condicional em caso de FP falso

Move o registrador da CPU `rs` para o registrador `rd` se o flag de código de condição da FPU número *cc* for 0. Se *cc* for omitido da instrução, o flag de código de condição 0 será assumido.

Move condicional em caso de FP verdadeiro

Move o registrador da CPU `rs` para o registrador `rd` se o flag de código de condição da FPU número *cc* for 1. Se *cc* for omitido da instrução, o bit de código de condição 0 é assumido.

Instruções de ponto flutuante

O MIPS possui um co-processador de ponto flutuante (número 1) que opera sobre números de ponto flutuante de precisão simples (32 bits) e precisão dupla (64 bits). Esse co-processador tem seus próprios registradores, que são numerados de `$f0` a `$f31`. Como esses registradores possuem apenas 32 bits, dois deles são necessários para manter doubles, de modo que somente registradores de ponto flutuante com números pares podem manter valores de precisão dupla. O co-processador de ponto flutuante também possui 8 flags de código de condição (cc), numerados de 0 a 7, que são alterados por instruções de comparação e testados por instruções de desvio (`bc1f` ou `bc1t`) e instruções move condicionais.

Os valores são movidos para dentro e para fora desses registradores uma word (32 bits) de cada vez pelas instruções `lwc1`, swc1, mtc1 e `mfc1` ou um double (64 bits) de cada vez por `ldc1` e `sdc1`, descritos anteriormente, ou pela pseudo-instruções `l.s`, `l.d`, `s.s` e `s.d`, descritas a seguir.

Nas instruções reais a seguir, os bits 21-26 são 0 para precisão simples e 1 para precisão double. Nas pseudo-instruções a seguir, `fdest` é um registrador de ponto flutuante (por exemplo, `$f2`).

Valor absoluto de ponto flutuante double

`abs.d fd, fs`

0x11	1	0	fs	fd	5
6	5	5	5	5	6

Valor absoluto de ponto flutuante single

`abs.s fd, fs`

0x11	0	0	fs	fd	5

Calcula o valor absoluto do double (single) de ponto flutuante no registrador `fs` e o coloca no registrador `fd`.

Adição de ponto flutuante double

`add.d fd, fs, ft`

0x11	0x11	ft	fs	fd	0
6	5	5	5	5	6

Adição de ponto flutuante single

`add.s fd, fs, ft`

0x11	0x10	ft	fs	fd	0
6	5	5	5	5	6

Calcula a soma dos doubles (singles) de ponto flutuante nos registradores `fs` e `ft` e a coloca no registrador `fd`.

Teto de ponto flutuante para word

`ceil.w.d fd, fs`

0x11	0x11	0	fs	fd	0xe
6	5	5	5	5	6

`ceil.w.s fd, fs`

0x11	0x10	0	fs	fd	0xe

Calcula o teto do double (single) de ponto flutuante no registrador `fs`, converte para um valor de ponto fixo de 32 bits e coloca a word resultante no registrador `fd`.

Comparação igual double

```
c.eq.d cc fs, ft
```

Comparação igual single

```
c.eq.s cc fs, ft
```

Compara o double (single) de ponto flutuante no registrador `fs` com aquele em `ft` e coloca o flag de condição de ponto flutuante *cc* em 1 se forem iguais. Se *cc* for omitido, o flag de código de condição 0 é assumido.

Comparação menor ou igual double

```
c.le.d cc fs, ft
```

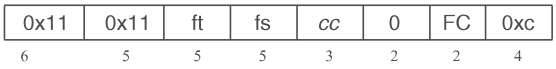
Comparação menor ou igual single

```
c.le.s cc fs, ft
```

Compara o double (single) de ponto flutuante no registrador `fs` com aquele no `ft` e coloca o flag de condição de ponto flutuante *cc* em 1 se o primeiro for menor ou igual ao segundo. Se o *cc* for omitido, o flag de código de condição 0 é assumido.

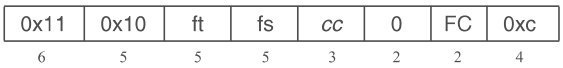
Comparação menor que double

```
c.lt.d cc fs, ft
```



Comparação menor que single

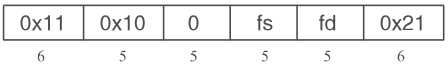
```
c.lt.s cc fs, ft
```



Compara o double (single) de ponto flutuante no registrador `fs` com aquele no `ft` e coloca o flag de condição de ponto flutuante *cc* em 1 se o primeiro for menor que o segundo. Se o *cc* for omitido, o flag de código de condição 0 é assumido.

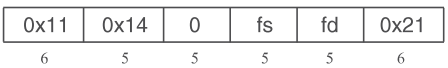
Converte single para double

```
cvt.d.s fd, fs
```



Converte integer para double

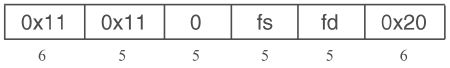
```
cvt.d.w fd, fs
```



Converte o número de ponto flutuante de precisão simples ou inteiro no registrador `fs` para um número de precisão dupla (simples) e o coloca no registrador `fd`.

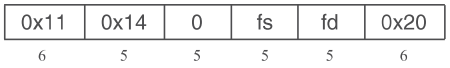
Converte double para single

```
cvt.s.d fd, fs
```



Converte integer para single

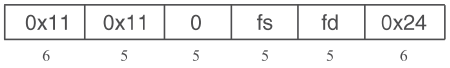
```
cvt.s.w fd, fs
```



Converte o número de ponto flutuante de precisão dupla ou inteiro no registrador `fs` para um número de precisão simples e o coloca no registrador `fd`.

Converte double para integer

```
cvt.w.d fd, fs
```



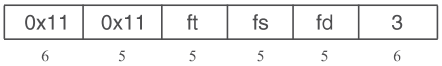
Converte single para integer

```
cvt.w.s fd, fs
```

Converte o número de ponto flutuante de precisão dupla ou simples no registrador `fs` para um inteiro e o coloca no registrador `fd`.

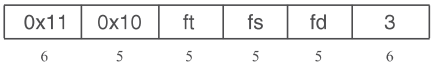
Divisão de ponto flutuante double

```
div.d fd, fs, ft
```



Divisão de ponto flutuante single

```
div.s fd, fs, ft
```



Calcula o quociente dos números de ponto flutuante de precisão dupla (simples) nos registradores `fs` e `ft` e o coloca no registrador `fd`.

Piso de ponto flutuante para word

```
floor.w.d fd, fs
```


```
floor.w.s fd, fs
```

Calcula o piso do número de ponto flutuante de precisão dupla (simples) no registrador `fs` e coloca a word resultante no registrador `fd`.

Carrega double de ponto flutuante

```
l.d fdest, address        pseudo-instrução
```

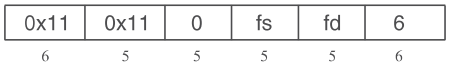
Carrega single de ponto flutuante

`l.s fdest, address` *pseudo-instrução*

Carrega o número de ponto flutuante de precisão dupla (simples) em `address` para o registrador `fdest`.

Move ponto flutuante double

`mov.d fd, fs`



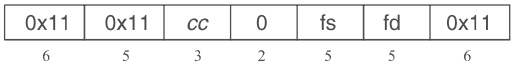
Move ponto flutuante single

`mov.s fd, fs`

Move o número de ponto flutuante de precisão dupla (simples) do registrador `fs` para o registrador `fd`.

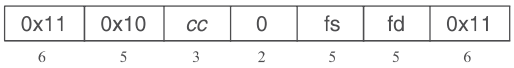
Move condicional de ponto flutuante double se falso

`movf.d fd, fs, cc`



Move condicional de ponto flutuante single se falso

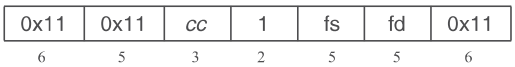
`movf.s fd, fs, cc`



Move o número de ponto flutuante de precisão dupla (simples) do registrador `fs` para o registrador `fd` se o flag do código de condição *cc* for 0. Se o *cc* for omitido, o flag de código de condição 0 é assumido.

Move condicional de ponto flutuante double se verdadeiro

`movt.d fd, fs, cc`



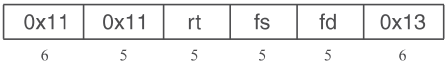
Move condicional de ponto flutuante single se verdadeiro

`movt.s fd, fs, cc`

Move o double (single) de ponto flutuante do registrador `fs` para o registrador `fd` se o flag do código de condição *cc* for 1. Se o *cc* for omitido, o flag do código de condição 0 será assumido.

Move ponto flutuante double condicional se não for zero

`movn.d fd, fs, rt`



Move ponto flutuante single condicional se não for zero

`movn.s fd, fs, rt`

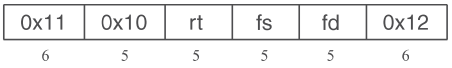
Move o número de ponto flutuante double (single) do registrador `fs` para o registrador `fd` se o registrador `rt` do processador não for 0.

Move ponto flutuante double condicional se for zero

```
movz.d fd, fs, rt
```

Move ponto flutuante single condicional se for zero

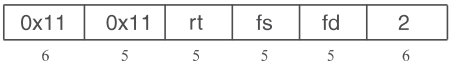
```
movz.s fd, fs, rt
```



Move o número de ponto flutuante double (single) do registrador `fs` para o registrador `fd` se o registrador `rt` do processador for 0.

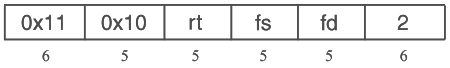
Multiplicação de ponto flutuante double

```
mul.d fd, fs, ft
```



Multiplicação de ponto flutuante single

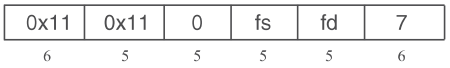
```
mul.s fd, fs, ft
```



Calcula o produto dos números de ponto flutuante double (single) nos registradores `fs` e `ft` e o coloca no registrador `fd`.

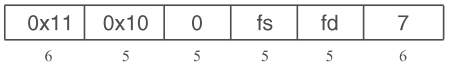
Negação double

```
neg.d fd, fs
```



Negação single

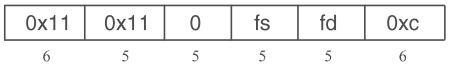
```
neg.s fd, fs
```



Nega o número de ponto flutuante double (single) no registrador `fs` e o coloca no registrador `fd`.

Arredondamento de ponto flutuante para word

```
round.w.d fd, fs
```




```
round.w.s fd, fs
```

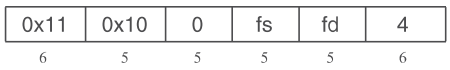
Arredonda o valor de ponto flutuante double (single) no registrador `fs`, converte para um valor de ponto fixo de 32 bits e coloca a word resultante no registrador `fd`.

Raiz quadrada de double

```
sqrt.d fd, fs
```

Raiz quadrada de single

```
sqrt.s fd, fs
```



Calcula a raiz quadrada do número de ponto flutuante double (single) no registrador `fs` e a coloca no registrador `fd`.

Store de ponto flutuante double

```
s.d fdest, address
```
pseudo-instrução

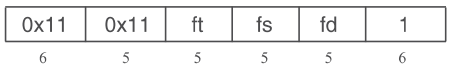
Store de ponto flutuante single

```
s.s fdest, address
```
pseudo-instrução

Armazena o número de ponto flutuante double (single) no registrador `fdest` em *address*.

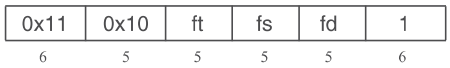
Subtração de ponto flutuante double

```
sub.d fd, fs, ft
```



Subtração de ponto flutuante single

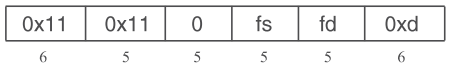
```
sub.s fd, fs, ft
```



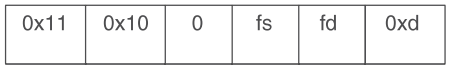
Calcula a diferença dos números de ponto flutuante double (single) nos registradores `fs` e `ft` e a coloca no registrador `fd`.

Truncamento de ponto flutuante para word

```
trunc.w.d fd, fs
```




```
trunc.w.s fd, fs
```



Trunca o valor de ponto flutuante double (single) no registrador `fs`, converte para um valor de ponto fixo de 32 bits e coloca a word resultante no registrador `fd`.

Instruções de exceção e interrupção

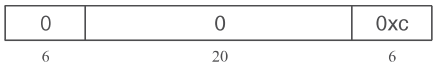
Retorno de exceção

```
eret
```

Coloca em 0 o bit EXL no registrador Status do co-processador 0 e retorna à instrução apontada pelo registrador EPC do co-processador 0.

Chamada ao sistema

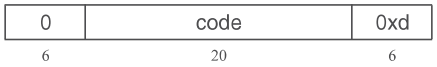
```
syscall
```



O registrador `$v0` contém o número da chamada ao sistema (ver Figura A.9.1) fornecido pelo SPIM.

Break

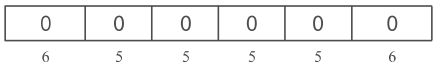
```
break code
```



Causa a exceção *código*. A Exceção 1 é reservada para o depurador.

Nop

```
nop
```



Não faz nada.

A.11 Comentários finais

A programação em assembly exige que um programador escolha entre os recursos úteis das linguagens de alto nível – como estruturas de dados, verificação de tipo e construções de controle – e o controle completo sobre as instruções que um computador executa. Restrições externas sobre algumas aplicações, como o tempo de resposta ou o tamanho do programa, exigem que um programador preste muita atenção a cada instrução. No entanto, o custo desse nível de atenção são programas em assembly maiores, mais demorados para escrever e mais difícil de manter do que os programas em linguagem de alto nível.

Além do mais, três tendências estão reduzindo a necessidade de escrever programas em assembly. A primeira tendência é em direção à melhoria dos compiladores. Os compiladores modernos produzem código comparável ao melhor código escrito manualmente – e, às vezes, melhor ainda. A segunda tendência é a introdução de novos processadores, que não apenas são mais rápidos, mas, no caso de processadores que executam várias instruções ao mesmo tempo, também mais difíceis de programar manualmente. Além disso, a rápida evolução dos computadores modernos favorece os programas em linguagem de alto nível que não estejam presos a uma única arquitetura. Finalmente, temos testemunhado uma tendência em direção a aplicações cada vez mais complexas, caracterizadas por interfaces gráficas complexas e muito mais recursos do que seus predecessores. Grandes aplicações são escritas por equipes de programadores e exigem recursos de modularidade e verificação semântica fornecidos pelas linguagens de alto nível.

Leitura adicional

Aho, A., R. Sethi e J. Ullman [1985]. *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley.

Ligeiramente desatualizado e faltando a cobertura das arquiteturas modernas, mas ainda é a referência padrão sobre compiladores.

Sweetman, D. [1999]. *See MIPS Run*, San Francisco CA: Morgan Kaufmann Publishers.