

Linguagens de montagem

Capítulo 5 – Instruções aritméticas e desvios

Ricardo Anido
Instituto de Computação
Unicamp

ADD							
Adição							
Syntax	Operação	Flags	Codificação				
<code>add rd, expr8</code>	$rd \leftarrow rd + \text{ext}(imd8)$	CNVZ	<div style="display: flex; justify-content: space-between; align-items: center;"> 31 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">0x10</td> <td style="width: 15%;">imd8</td> <td style="width: 15%;">rd</td> <td style="width: 15%;">-</td> </tr> </table>	0x10	imd8	rd	-
0x10	imd8	rd	-				
<code>add rd, rf</code>	$rd \leftarrow rd + rf$	CNVZ	<div style="display: flex; justify-content: space-between; align-items: center;"> 31 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">0x11</td> <td style="width: 15%;">-</td> <td style="width: 15%;">rd</td> <td style="width: 15%;">rf</td> </tr> </table>	0x11	-	rd	rf
0x11	-	rd	rf				

- ▶ armazenam o estado resultante de algumas instruções do processador.
- ▶ Nem todas as instruções afetam todos os bits de estado; por exemplo, instruções de transferência de dados, como MOV ou LD, não afetam nenhum bit de estado.
- ▶ São armazenados em um registrador de estado, também chamado *flags*.

- ▶ C: vai-um (*carry*). Ligado (ou seja, tem valor 1) se a operação causou vai-um (*carry-out*) ou empresta-um (*carry-in*), desligado caso contrário.
- ▶ Z: zero. Ligado se o resultado foi zero, desligado caso contrário.
- ▶ N: sinal. Cópia do bit mais significativo do resultado; considerando aritmética com sinal, se N igual a zero, o resultado é maior ou igual a zero. Se N igual a 1, resultado é negativo.
- ▶ V: estouro de campo (*overflow*), para operações com números com sinal em complemento de dois. Ligado se ocorreu estouro de campo, desligado caso contrário. Calculado como o ou-exclusivo entre o vai-um do bit mais significativo do resultado e o vai-um do segundo bit mais significativo do resultado.

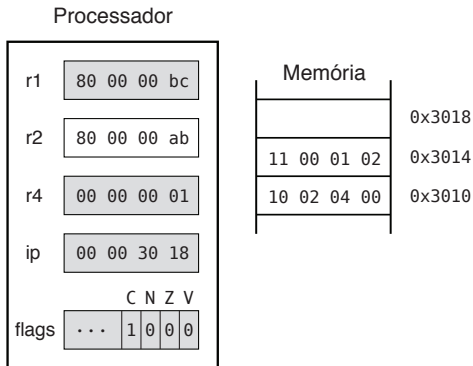
Adição

|@ exemplos de instrução add, com endereçamento
|@ imediato e entre registradores

|

```
00003000 [01 11 01 00] | set r1,0x11 @ carrega alguns valores
00003004 [02 00 02 00] | set r2,0x800000ab @ nos registradores para
           [80 00 00 ab] | @ ilustrar a operação
0000300c [01 ff 04 00] | set r4,-1 @ da instrução add
00003010 [11 00 01 02] | add r1,r2 @ adição, registradores
00003014 [10 02 04 00] | add r4,2 @ adição, valor imediato
```

Subtração



Subtração

Subtração													
Syntax	Operação	Flags	Codificação										
<code>SUB rd, expr8</code>	$rd \leftarrow rd - \text{ext}(imd8)$	CNVZ	<table border="1"><tr><td>31</td><td></td><td></td><td></td><td>0</td></tr><tr><td>0x12</td><td><i>imd8</i></td><td><i>rd</i></td><td>-</td><td></td></tr></table>	31				0	0x12	<i>imd8</i>	<i>rd</i>	-	
31				0									
0x12	<i>imd8</i>	<i>rd</i>	-										
<code>SUB rd, rf</code>	$rd \leftarrow rd - rf$	CNVZ	<table border="1"><tr><td>31</td><td></td><td></td><td></td><td>0</td></tr><tr><td>0x13</td><td>-</td><td><i>rd</i></td><td><i>rf</i></td><td></td></tr></table>	31				0	0x13	-	<i>rd</i>	<i>rf</i>	
31				0									
0x13	-	<i>rd</i>	<i>rf</i>										

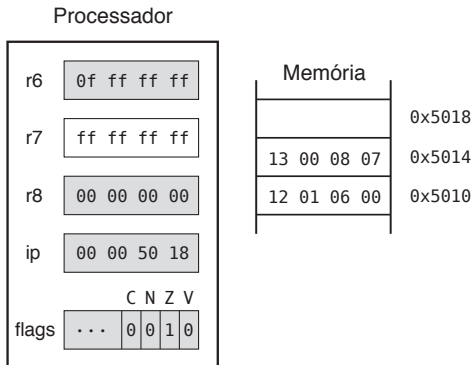
Subtração

|@ exemplos de instrução sub com endereçamento

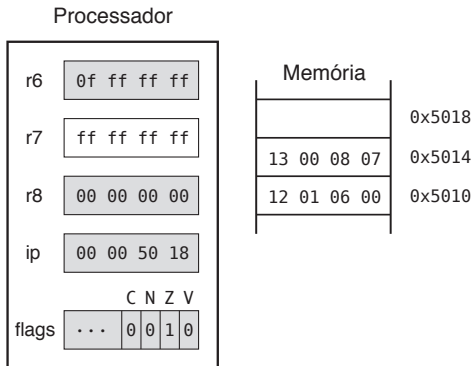
|@ imediato e entre registradores

```
00005000 [02 00 06 00] | set r6,0x10000000 @ carrega alguns valores
00005004 [10 00 00 00] | @ em registradores para
00005008 [01 ff 07 00] | set r7,-1 @ ilustrar a operação
0000500c [01 ff 08 00] | set r8,-1 @ da instrução sub
00050010 [12 01 06 00] | sub r6,1 @ sub com ender. imediato
00050014 [13 00 08 07] | sub r8,r7 @ sub entre registradores
```


Subtração



Subtração



Exemplo

```
int a, b, c;  
...  
    a = b + c - 2;  
...
```

Exemplo

```
@ reserva espaço para as variáveis inteiras a, b e c
a:  .skip 4
b:  .skip 4
c:  .skip 4
...
ld   r0,b      @ r0 e r1 são usados como auxiliares
ld   r1,c      @ para armazenar as variáveis
add  r0,r1     @ r0 agora tem b+c
sub  r0,2      @ r0 agora tem b+c-2
st   a,r0      @ e armazena o resultado na variável a
```

Instruções de desvio alteram o valor do registrador interno `ip`, de forma que podemos controlar o fluxo de execução do programa.

- ▶ Podem ser *incondicionais* ou *condicionais*.

Desvio incondicional

- ▶ A instrução de desvio incondicional tem apenas um operando e seu funcionamento é bastante simples: o valor do operando, chamado de *endereço alvo* do desvio, é copiado para o registrador *ip*.
- ▶ Assim, a próxima instrução a ser executada após a instrução de desvio incondicional é a armazenada no endereço alvo.

Desvio incondicional

Desvio incondicional																							
Syntax	Operação	Flags	Codificação																				
<code>jmp expr32</code>	$ip \leftarrow imd32$	-	<table border="1"><tr><td>31</td><td></td><td></td><td></td><td>0</td></tr><tr><td>0x20</td><td>-</td><td>-</td><td>-</td><td></td></tr><tr><td>31</td><td></td><td></td><td></td><td>0</td></tr><tr><td colspan="5"><i>imd32</i></td></tr></table>	31				0	0x20	-	-	-		31				0	<i>imd32</i>				
31				0																			
0x20	-	-	-																				
31				0																			
<i>imd32</i>																							
<code>jmp rd</code>	$ip \leftarrow rd$	-	<table border="1"><tr><td>31</td><td></td><td></td><td></td><td>0</td></tr><tr><td>0x21</td><td>-</td><td><i>rd</i></td><td>-</td><td></td></tr></table>	31				0	0x21	-	<i>rd</i>	-											
31				0																			
0x21	-	<i>rd</i>	-																				

Desvio incondicional, endereçamento imediato

```
                                |@ exemplo de instrução jmp, ender. imediato
                                |   .org 0x200
00000200 [01 00 00 00] |   set  r0,0      @ uma instrução qualquer
00000204 [20 00 00 00] |   jmp  fora     @ instrução de desvio
           [00 00 30 00] |                   @ incondicional
0000020c [00 00 01 02] |   mov  r1,r2    @ esta instrução não é
                                |                   @ executada
                                |...
                                |   .org 0x3000
                                | fora:
00003000 [00 00 02 01] |   mov  r2,r1    @ esta instrução é a
                                |                   @ próxima a ser executada
                                |                   @ após o desvio
```


Desvio incondicional, endereçamento por registrador

```
                                |@ exemplo de instr. jmp, por registrador
                                |...
00000208 [21 00 0a 00] |      jmp  r10      @ instrução de desvio
0000020c [00 00 03 04] |      mov  r3,r4    @ esta instrução não é
                                |                          @ executada
                                | ...
                                |      .org 0x4000
                                | depois:
00004000 [00 00 04 03] |      mov  r4,r3    @ se r10 tem o valor
                                |                          @ 0x4000, esta é a próxima
                                |                          @ instrução executada
                                |                          @ após o desvio da linha 5
```

- ▶ Instruções de desvio condicional executam ou não o desvio dependendo do valor de um ou mais de bits de estado (que como já vimos são alterados pela execução de algumas instruções)
- ▶ Os comandos em linguagem de montagem correspondentes às instruções de desvio têm os nomes no formato *Jcond*, ou seja, a letra J seguida de um indicativo de condição *cond*, como JNC ou JZ.

Desvios condicionais

<i>Jcond</i>											
Desvio Condicional											
Sintaxe	Operação	Flags	Codificação								
<i>Jcond expr32</i>	$ip \leftarrow ip + ext32(imd8)$	-	<table border="1"><tr><td>31</td><td></td><td></td><td>0</td></tr><tr><td>0xcc</td><td><i>imd8</i></td><td>-</td><td>-</td></tr></table>	31			0	0xcc	<i>imd8</i>	-	-
31			0								
0xcc	<i>imd8</i>	-	-								

Desvios condicionais

Instr.	Nome	Condição	Codif.
jc	desvia se vai-um (menor sem sinal)	C=1	0x22
jnc	desvia se não vai-um (maior ou igual sem sinal)	C=0	0x23
jz	desvia se zero	Z=1	0x24
jnz	desvia se diferente de zero	Z=0	0x25
jo	desvia se overflow	V=1	0x26
jno	desvia se não overflow	V=0	0x27

Desvios condicionais (cont.)

Instr.	Nome	Condição	Codif.
js	desvia se sinal igual a um	$N=1$	0x28
jns	desvia se sinal igual a zero	$N=0$	0x29
j1	desvia se menor (com sinal)	$N \neq V$	0x2a
jle	desvia se menor ou igual (com sinal)	$Z=1$ ou $N \neq V$	0x2b
jg	desvia se maior (com sinal)	$Z=0$ e $N=V$	0x2c
jge	desvia se maior ou igual (com sinal)	$N=V$	0x2e
ja	desvia se acima (maior sem sinal)	$C=0$ e $Z=0$	0x2f
jna	desvia se não acima (menor ou igual sem sinal)	$C=1$ ou $Z=1$	0x1f

Exemplo de desvios condicionais

```

| @ exemplo de desvio condicional
|   .org 0x500
00000500 [12 01 08 00] |   sub  r8,1      @ uma instrução montada
|                                     @ no endereço 0x500
00000504 [24 04 00 00] |   jz   continua @ um desvio condicional
00000508 [01 64 08 00] |   set  r8,100   @ esta instrução é executada
|                                     @ se resultado da instrução
|                                     @ sub for diferente de zero
|
| continua:
0000050c [00 00 05 06] |   mov  r5,r6    @ próxima instrução a
|                                     @ ser executada após jz
|                                     @ se o resultado de sub
|                                     @ for igual a zero
```

- ▶ A instrução JZ é usada para testar igualdade, e obviamente pode ser utilizada para comparações tanto entre valores inteiros com sinal como entre valores inteiros sem sinal.
- ▶ As instruções JL, JLE, JG e JGE devem ser usadas na comparação números inteiros com sinal.
- ▶ As instruções JC, JNC, JA e JNA devem ser usadas na comparação de números inteiros sem sinal.

Problema

Escreva um trecho de programa que, dados dois números inteiros sem sinal em `r1` e `r2`, coloque em `r1` o menor valor e em `r2` o maior valor.

ordena:

```
mov  r0,r1          @ vamos usar r0 como rascunho
sub  r0,r2          @ valor de r1 é maior que r2?
jna  ordena_final   @ não é --- nada a fazer
mov  r0,r1          @ troca r1 com r2
mov  r1,r2          @ usando r0
mov  r2,r0          @ como temporário
```

ordena_final:

Problema

Escreva um trecho de programa que coloque no registrador `r0` o maior valor entre `r1`, `r2` e `r3`. Suponha que os registradores contenham números inteiros com sinal.

Solução

```
maior:
    mov    r4,r1        @ usando r4 como rascunho
    sub    r4,r2        @ compara r1 e r2
    jge    um          @ desvia se r1 maior
    mov    r4,r2        @ guarda maior valor em r4
    mov    r0,r2        @ e em r0
    jmp    outro

um:
    mov    r4,r1        @ guarda maior valor em r4
    mov    r0,r1        @ e em r0

outro:
    sub    r4,r3        @ compara maior entre r1 e r2 com r3
    jge    final       @ r3 é menor, r0 já tem maior valor
    mov    r0,r3        @ maior é r3, atualiza r0

final:
```

Nova instrução: comparação

CMP											
Comparação											
Syntax	Operação	Flags	Codificação								
$CMP\ rd, expr8$	$rd - ext(imd8)$	CNVZ	<table border="1"><tr><td>31</td><td></td><td></td><td>0</td></tr><tr><td>0x14</td><td><i>imd8</i></td><td><i>rd</i></td><td>-</td></tr></table>	31			0	0x14	<i>imd8</i>	<i>rd</i>	-
31			0								
0x14	<i>imd8</i>	<i>rd</i>	-								
$CMP\ rd, rf$	$rd - rf$	CNVZ	<table border="1"><tr><td>31</td><td></td><td></td><td>0</td></tr><tr><td>0x15</td><td>-</td><td><i>rd</i></td><td><i>rf</i></td></tr></table>	31			0	0x15	-	<i>rd</i>	<i>rf</i>
31			0								
0x15	-	<i>rd</i>	<i>rf</i>								

Solução com instrução CMP

```
maior:
    mov  r0,r1      @ assume que r1 é o maior
    cmp  r0,r2      @ compara r1 e r2
    jge  outro     @ desvia se r1 maior
    mov  r0,r2      @ r2 é maior, guarda em r0
outro:
    cmp  r0,r3      @ compara maior entre r1 e r2 com r3
    jge  final     @ r0 já tem o maior valor
    mov  r0,r3      @ maior é r3, atualiza r0
final:
    ...           @ final do trecho
```

Revisão

```
a:  .skip 4
b:  .skip 4

    .org 0x100
c:  .word -1
d:  .byte 'abcd'

    .org 0x200
init:
    set r10,a
    set r11,b
    set r4,c
    ld  r5,c
    jmp init2
    ld  r6,[r4]
    ldb r7,c
    ld  r8,d

init2:
    st  d,r4
```

Problema

Traduza o trecho de programa escrito em C a seguir, que contém um comando *if* que testa a igualdade de duas variáveis inteiras, para linguagem de montagem do LEG.

```
int a, b;

...
    if (a==b)
        a=a+b;
    else
        b=a+b;
...
```

Solução

@ aloca variáveis inteiras
@ note que podem armazenar valores com ou sem sinal
@ o "tipo" deve ser interpretado pelo programador

```
a:  .skip 4  
b:  .skip 4  
...
```

```
if:  
    ld    r0,a           @ if (a==b)  
    ld    r1,b  
    cmp   r0,r1         @ compara a com b  
    jnz   else  
    add   r0,r1         @ a=a+b  
    st    a,r0  
    jmp   final_if  
else:  
    add   r0,r1         @ b=a+b  
    st    b,r1  
final_if:
```


Problema

Traduza para linguagem de montagem LEG o trecho de programa em C a seguir, que inclui um comando *for*.

```
int i,a,b;
```

```
...
```

```
    for (i=0;i<100;i++) {  
        a=a+b;  
    }
```

```
...
```

Solução

```
@ aloca variáveis inteiras
i:   .skip 4
a:   .skip 4
b:   .skip 4
    ...
inicio_for:
    set  r0,0
    st   i,r0           @ i=0
teste_for:
    ld   r0,i           @ verifica se executa bloco de comandos do for
    cmp  r0,100        @
    jge  final_for     @ i>=100? caso verdadeiro, finaliza
corpo_for:
    ld   r1,a           @ bloco de comandos do for tem apenas um
    ld   r2,b           @ comando de atribuição
    add  r1,r2
    st   a,r1           @ a=a+b
incremento:
    add  r0,1           @ i++
    st   i, r0
```

Solução com otimizações

```
@ aloca variáveis inteiras
i:  .skip 4
a:  .skip 4
b:  .skip 4
    ...
inicio_for:
    set   r0,0           @ r0 vai conter i
    ld   r1,a           @ prepara para os registradores
    ld   r2,b           @ para o corpo do comando for
teste_for:
    cmp   r0,100        @ verifica se executa bloco de comandos do for
    jge  final_for      @ i>=100? caso verdadeiro, finaliza
corpo_for:
    add  r1,r2
    add  r0,1           @ i++
    jmp  teste_for      @ terminou o bloco, executa o teste
final_for:
    st   i,r0           @ atualiza valor de i
    st   a,r1           @ atualiza valor de a
    ...
```

Desvios condicionais não podem desviar para “longe”

Se `muito_longe` estiver “muito distante” do desvio condicional, a distância relativa não pode ser representada no campo *imd8* da instrução JNC:

```
    sub  r2,r1
    jnc  muito_longe
    add  r0,r3
    ...
muito_longe:
    ...
```

O montador indicará um erro, informando que não é possível montar o programa.

Desvios condicionais não podem desviar para “longe”

Solução: inverter a lógica do desvio condicional e usar, adicionalmente, um desvio incondicional:

```
    sub  r2,r1
    jc   bem_perto
    jmp  muito_longe
bem_perto:
    add  r0,r3
    ...
muito_longe:
    ...
```

Este é praticamente o único caso em que dois comandos de desvios precisam ser usados em sequência; normalmente dois desvios consecutivos indicam uma construção errada.

Escreva um trecho de programa para determinar qual o maior valor de um vetor de números inteiros de 32 bits, sem sinal, cujo endereço inicial é dado em $r2$. Considere que todos os números são distintos. Inicialmente $r3$ contém o número de elementos presentes na vetor; suponha que $r3 \neq 0$. Ao final do trecho, $r0$ deve conter o valor máximo e $r1$ deve conter o endereço do valor máximo.

Solução

```
início:
    mov  r1,r2      @ guarda apontador para início do vetor
    ld  r0,[r1]    @ e valor máximo corrente (o primeiro valor)
proximo:
    add  r2,4      @ avança ponteiro para próximo elemento
    sub  r3,1      @ um elemento a mais verificado
    jz   final     @ terminou de verificar todo o vetor?
    ld  r4,[r2]    @ não, então toma mais um valor
    cmp  r0,r4     @ compara com máximo corrente
    jnc  proximo   @ desvia se menor ou igual
    mov  r1,r2     @ achamos um maior, guarda novo apontador
    mov  r0,r4     @ e novo máximo
    jmp  proximo   @ e continua a percorrer o vetor
final:
    ...          @ final do trecho
```

Problema

Traduza o trecho de programa em C a seguir, que contém um comando *switch*, para a linguagem de montagem do LEG.

```
switch(val) {  
    case 1000:  
        x = y; break;  
    case 1001:  
        y = x; break;  
    case 1004:  
        t = x;  
    case 1005:  
        x = 0; break;  
    default:  
        x = y = t = 0;  
}
```


Solução

@ tradução de comando switch, versão sequencial

@ comando switch (val)

switch:

```
    ld   r0,val           @ carrega variável de seleção 'val'
```

case1000:

```
    set  r1,1000         @ primeira seleção
    cmp  r0,r1           @ verifica se igual
    jnz  case1001        @ desvia se diferente
    ld   r0,y
    st   x,r0            @ x = y
    jmp  final           @ break
```

case1001:

```
    set  r1,1001        @ segunda seleção
    cmp  r0,r1           @ verifica se igual
    jnz  case1004        @ desvia se diferente
    ld   r0,x
    st   y,r0            @ y = x
    jmp  final           @ break
```

Solução

case1004:

```
set  r1,1004      @ terceira seleção
cmp  r0,r1        @ verifica se igual
jnz  case1005     @ desvia se diferente
ld   r0,x
st   t,r0         @ t = x
jmp  sembreak     @ note que não há break, executa também o bloco 'case
```

case1005:

```
set  r1,1005     @ quarta seleção
cmp  r0,r1        @ verifica se igual
jnz  default     @ desvia se diferente
```

sembreak:

```
set  r0,0
st   x,r0        @ x = 0
jmp  final       @ break
```

default:

```
set  r0,0
st   t,r0        @ t = 0
st   x,r0        @ x = 0
st   y,r0        @ y = 0
```

final: