4 O processador Faíska

Neste capítulo iniciaremos o estudo mais detalhado do repertório de instruções do processador Faíska, ilustrando com exemplos de trechos de programas o uso das diversas instruções. As instruções serão apresentadas usando um *descritor* no formato de uma tabela, como o mostrado na Figura 4.1, contendo o mnemônico da instrução (MNE na figura), o nome, o formato em linguagem de montagem, a operação descrita de forma sucinta, e a codificação descrita de forma esquemática. O descritor inclui também um campo *Flags*, cujo significado será introduzido na Seção 4.3.

MNE	Nome da instrução			
	Formato	Operação	Flags	Codificação

Fig. 4.1 Formato do descritor de instruções.

As instruções serão apresentadas em grupos: transferência de dados,

operações aritméticas, instruções para controle de fluxo e operações lógicas.

4.1 Transferência de dados

Instruções de transferência de dados são as que permitem a transferência de dados entre dois registradores do processador ou entre a memória e um registrador. Um exemplo de instrução de transferência de dados é a instrução “carrega registrador com valor imediato”, que já foi introduzida no capítulo anterior. Nesta seção iremos estudar instruções para transferência de dados que utilizam outros modos de endereçamento.

4.1.1 Transferência entre registradores

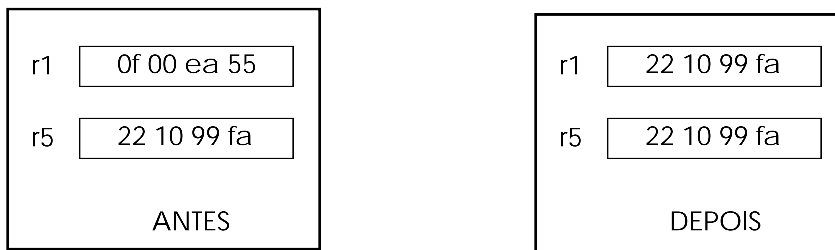
Os operandos para instruções de transferência entre registradores são sempre dois registradores: o registrador destino (operando mais à esquerda na linguagem de montagem) e o registrador fonte (operando mais à direita). A operação executada é óbvia: o valor do registrador fonte é copiado para o registrador destino.

Qualquer dos registradores de propósito geral do Faíska (r0 a r31) pode ser usado como registrador destino ou fonte. O exemplo 4.1 mostra um trecho de código e uma representação esquemática do estado do processador antes e após a execução do trecho de código (no esquema do processador são apresentados apenas os registradores envolvidos no exemplo).

MOV	Carrega registrador			
	Formato	Operação	Flags	Codificação
	ld rdst, rsrc	*rdst ← rsrc*	–	00 | – | rdst | rsrc

Exemplo 4.1: instrução tranferência entre registradores.

```
mov     r1, r5          ; exemplo de instrução mov
                        ; codificação desta instrução é 00000105h
```



4.1.2 Transferência da Memória para um registrador

O Faíska possui três instruções para transferência de valores da memória para um registrador. Cada uma destas instruções utiliza um modo de endereçamento diferente.

Modos de endereçamento

A maneira utilizada pelo processador para calcular o endereço, na memória, do operando de uma instrução é chamado *modo de endereçamento*. Nesta seção veremos três modos de endereçamento: imediato, direto e indireto por registrador; outros modos de endereçamento serão vistos mais adiante.

Endereçamento imediato

Este modo de endereçamento já foi introduzido no capítulo anterior, quando vimos a instrução carrega constante em registrador. Aqui vamos apenas estudar melhor como é feita a codificação desta instrução no Faíska. O Faíska possui duas instruções de carga de registrador com endereçamento imediato: uma em que a constante a ser carregada é codificada em apenas um byte; e uma em que a constante é codificada em 32 bits (esta última forma é a que foi apresentada no capítulo anterior).

No caso em que o operando é codificado em 32 bits, a instrução ocupa duas palavras. Para constantes que podem ser representadas em 8 bits, a instrução utiliza 8 bits (campo *imm8*) para armazenar a constante como um inteiro com sinal em complemento de dois. Assim, esta forma só permite carregar constantes menores que 127 e maiores que -128. Como veremos, na programação em linguagem de montagem muitas vezes utilizamos constantes dentro desse intervalo, o que nos permite utilizar a forma mais econômica com bastante freqüência. É importante notar que todos os 32 bits do registrador destino são carregados pela instrução *set*, mesmo na sua forma mais curta. Nesse caso, quando um valor *imm8* positivo é carregado em um registrador, os 24 bits mais significativos do registrador são zerados. Quando o campo *imm8* tem um valor negativo, os 24 bits mais significativos do registrador destino recebem o valor 1, de forma a fazer com que o registrador tenha o valor negativo correto. Ou seja, o bit de sinal do operando *imm8* é 'estendido' para os bits mais significativos do registrador destino.

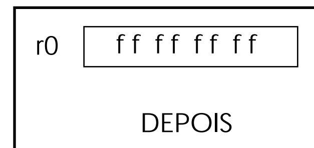
SET	Carrega registrador com valor imediato		
Formato	Operação	Flags	Codificação
set *rdst, label*	*rdst* ← *extend(imm8)*	–	01 | imm8 | rdst | –
set *rdst, label*	*rdst* ← *imm32*	–	02 | – | rdst | – imm32

O exemplo 4.2 mostra o estado do processador antes e depois da execução de uma instrução *set*. Ovalor 'x' que aparece como conteúdo inicial do registrador na figura será utilizado ao longo do texto para indicar que o valor do registrador naquele momento é sem importância

no contexto do exemplo.

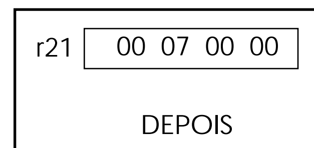
Exemplo 4.2: instrução de carga de registrador com endereçamento imediato.

```
        set     r0, -1          ; instrução ocupa apenas uma palavra
                                ; o código desta instrução é 01 ff 00 00
```



Exemplo 4.3: instrução de carga de registrador com endereçamento imediato.

```
        set     r21, var1       ; carrega endereço de var1
                                ; esta instrução ocupa duas palavras:
                                ; 02 00 15 00   e   00 07 00 00
        ...
.data
        org     70000h
var1:   dw      ?               ; uma variável com valor qualquer
                                ; montada no endereço 70000h
```



Endereçamento direto: o conceito de variáveis

O endereçamento imediato só serve para carregar o valor de uma constante em um registrador, já que o mesmo valor será carregado toda vez que uma determinada instrução *set* for executada. Para implementar o conceito de variáveis é necessário ler e escrever posições específicas da

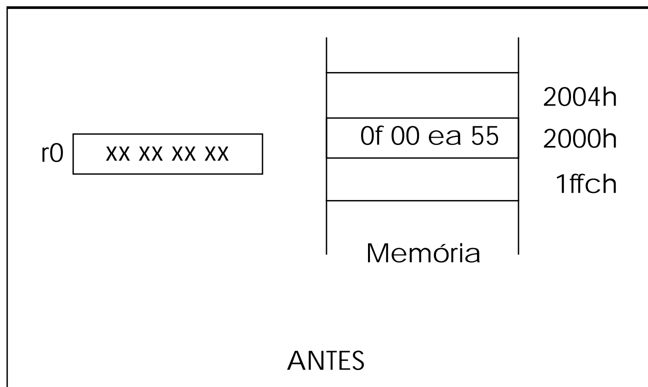
memória. Assim, uma variável (inteiro i, por exemplo) pode ser associada a uma determinada posição da memória (2000h, por exemplo); toda vez que se quiser acessar a variável *i*, faz-se um acesso à posição de memória correspondente.

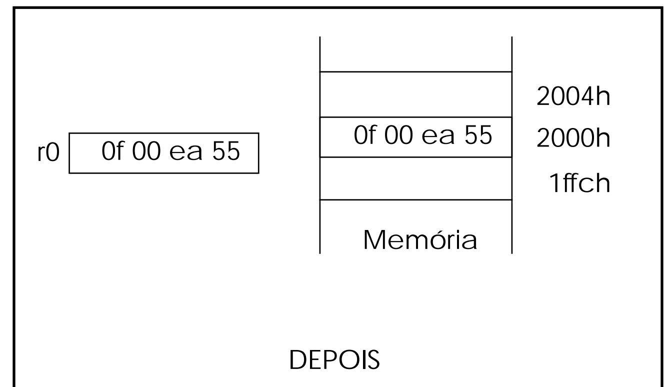
Uma maneira de acessar posições específicas é utilizar o chamado *modo de endereçamento direto*. Na instrução de carga com endereçamento imediato a própria instrução contém o valor a ser carregado. Já na instrução de carga com endereçamento direto a instrução contém o *endereço* da posição de memória que contém o valor a ser acessado. No Faíska, esta instrução é codificada em duas palavras, com a segunda palavra identificando o endereço do operando.

LD	Carrega registrador			
	Formato	Operação	Flags	Codificação
	ld rdst, label	*rdst* ← *mem[label]*	–	03 | – | rdst | – imm32

Exemplo 4.4: instrução de carga de registrador com endereçamento direto.

```
        ld      r0, cont        ; exemplo de instrução ld
                                ; esta instrução é codificada em duas palavras:
                                ; 03 00 00 00    e    00 00 20 00
        ...
.data
        org     2000h
cont:   dw      ?               ; uma variável com valor qualquer
                                ; montada no endereço 2000h
```





Endereçamento indireto por registrador

Nesta forma de endereçamento, um registrador contém o endereço do operando, ao invés deste ser fixo, codificado na instrução, como no endereçamento direto. Na linguagem de montagem, utilizaremos para

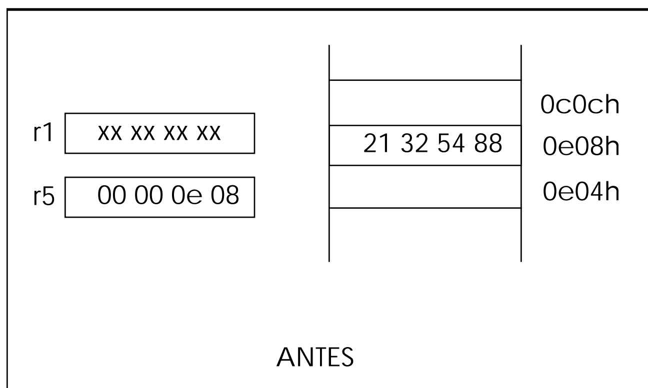
LD	Carrega registrador			
	Formato	Operação	Flags	Codificação
	ld rdst, (rsrc)	*rdst ← mem[rsrc]*	–	04 | – | rdst | rsrc

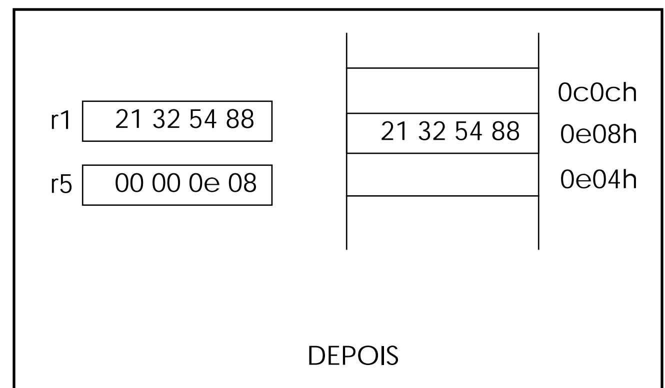
a instrução de carga de registrador com endereçamento indireto por registrador o mesmo mnemônico *ld* já utilizado anteriormente, mas indicaremos o modo de endereçamento distinto pela grafia do operando.

Na instrução *ld* com modo de endereçamento direto, o operando é um rótulo (no caso geral, qualquer expressão constante). No caso de instrução *ld* com modo de endereçamento indireto por registrador, o operando será sempre o nome de um registrador entre parênteses. Essa convenção de utilizar o mesmo mnemônico para duas instruções distintas é apenas uma conveniência para o programador, e dá a ilusão da existência de "uma" instrução *ld* com dois modos de endereçamento; é importante compreender no entanto que para o processador o que existe são duas instruções bastante distintas, com codificações diferentes.

Exemplo 4.5: instrução de carga de registrador com endereçamento indireto por registrador.

```
ld      r1, (r5)        ; exemplo de instrução ld com endereçamento
                        ; indireto por registrador
```





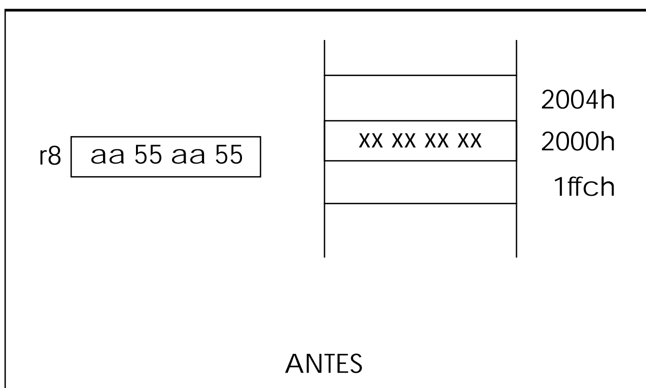
4.1.3 Transferência de um Registrador para a Memória

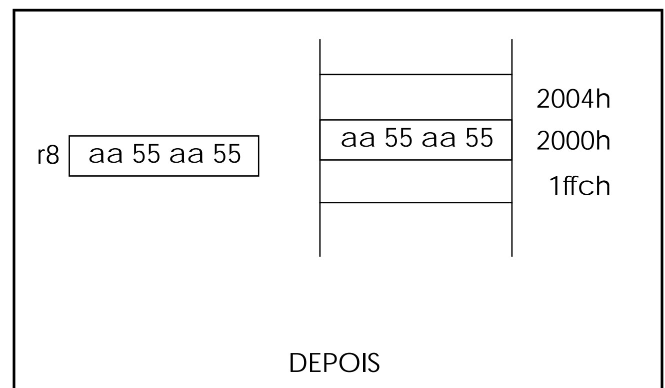
As instruções de armazenamento de um registrador na memória efetuam a operação inversa das instruções de carga de regsitrador. No Faíska, a instrução de armazenamento utiliza o mnemônica *st* (do inglês *store*) e possui duas variantes, correspondentes aos modos de endereçamento direto e indireto por registrador.

ST	Guarda registrador na memória			
	Formato	Operação	Flags	Codificação
	st *label, rsrc*	*mem[imm32] ← rsrc*	–	05 | – | – | rsrc imm32
	st *(rdst), rsrc*	*mem[rdst] ← rsrc*	–	06 | – | rdst | rscr

Exemplo 4.6:

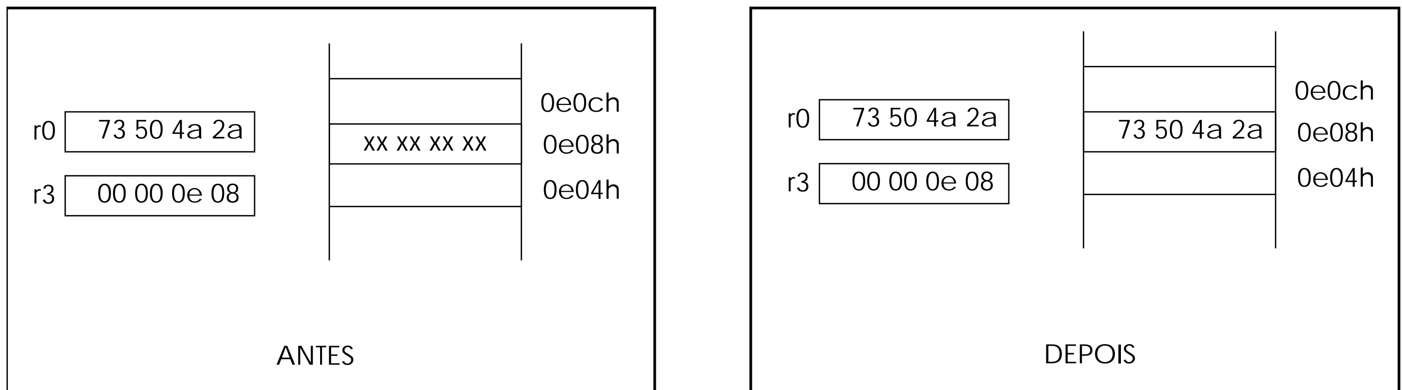
```
        st     flag, r8  ; exemplo de instrução st, endereçamento direto
        ...
.data
        org    2000h
flag:   dw     ?              ; uma variável com valor qualquer
                              ; montada no endereço 2000h
```





Exemplo 4.7:

```
        st     (r3), r0       ; exemplo de instrução st, endereçamento indireto
                              ; por registrador
```



4.1.4 Tranferência de bytes entre registradores e memória

Como é muito comum a necessidade de manipular quantidades de 8 bits (especialmente caracteres), o Faíska inclui também instruções para carregar e armazenar um byte da memória em registradores.

Carga de byte da memória para registrador

Nesta instrução, apenas um byte é carregado da memória para o registrador destino. No Faíska, o byte carregado é colocado nos 8 bits menos significativos do registrador destino; os 24 bits mais significativos do registrador são zerados.

LDB	Carrega byte em registrador			
	Formato	Operação	Flags	Codificação
	ldb *rdst, (rsrc)*	*rdst ← mem8[rsrc]*	-	07 – rdst rsrc
	ldb *rdst, label*	*rdst ←mem8[imm32]*	–	08 – rdst – imm32

Exemplo 4.8:

```
        ldb     r4, num         ; exemplo de instrução ldb, endereçamento direto
        ...
.data
        org     2007h
num:    dw      ?               ; uma variável com valor qualquer
                                ; montada no endereço 2007h
```

r4 xx xx xx xx

200bh
42 20 38 77 2007h
2003h

ANTES

r4 00 00 00 77

200bh
42 20 38 77 2007h
2003h

DEPOIS

Exemplo 4.9:

```
        ldb     r1, (r5)        ; exemplo de instrução ldb, endereçamento
                                ; indireto por registrador
```

r1 xx xx xx xx
r5 00 00 10 0a
100eh
8a 10 11 8e 100ah
1006h
ANTES

r1 00 00 00 8e
r5 00 0010 0a
100eh
4a 10 11 8e 100ah
1006h
DEPOIS

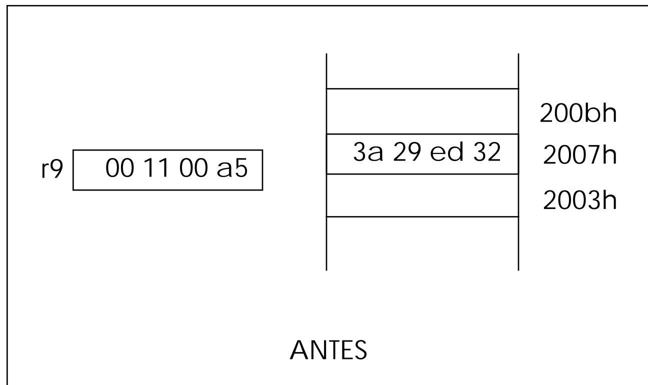
Armazenamento de byte de registrador em memória

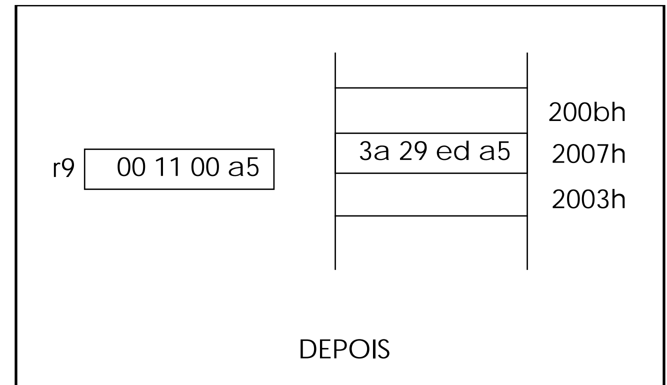
STB	Guarda byte na memória			
	Formato	Operação	Flags	Codificação
	stb *(rdst+ val8)*, *rsrc*	*mem8[rdst]* ← *rsrc8*	-	09 | – | rdst | rsrc
	stb *label*, *rsrc*	*mem8[imm32]* ← *rsrc8*	–	0A | – | – | rscr imm32

No Faíska, duas instruções de armazenamento de byte estão definidas, uma com endereçamento direto e uma com endereçamento indireto por registrador.

Exemplo 4.10:

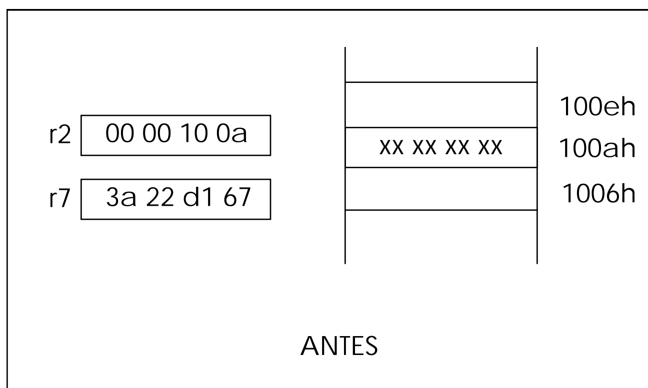
```
        stb     num, r9         ; exemplo de instrução stb, endereçamento direto
        ...
.data
        org     2007h
num:    dw      ?               ; uma variável com valor qualquer
                                ; montada no endereço 2007h
```

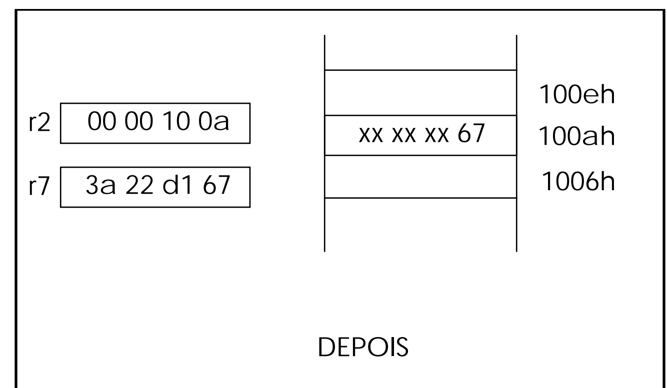




Exemplo 4.11:

```
stb    (r2), r7        ; exemplo de instrução stb
```





4.2 Instruções aritméticas

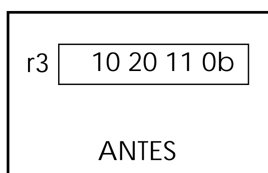
No grupo das instruções aritméticas encontramos instruções para somar, subtrair, multiplicar e dividir números inteiros. Nesta seção estudaremos apenas as instruções para somar e subtrair; multiplicação e divisão serão apresentadas mais adiante.

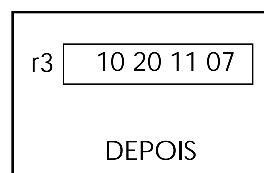
No Faíska as instruções de somas e subtrações têm sempre como operando destino um registrador; o segundo operando pode ser um outro registrador ou um valor imediato codificado em 8 bits (inteiro em complemento de dois).

ADD	Adição			
	Formato	Operação	Flags	Codificação
	add *rdst*, *val8*	*rdst* ← *rdst* + *ext(imm8)*	OCSZ	0B | imm8 | rdst | –
	add *rdst*, *rsrc*	*rdst* ← *rdst* + *rsrc*	OCSZ	0C | – | rdst | rsrc

Exemplo 4.12:

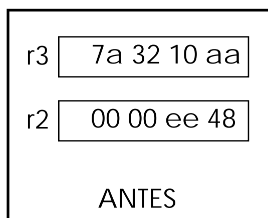
```
add    r3, -4        ; instrução soma com valor imediato
```

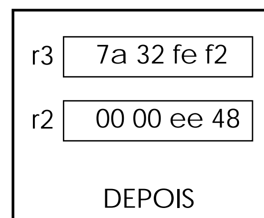




Exemplo 4.13:

```
add    r3, r2        ; exemplo de instrução soma
```

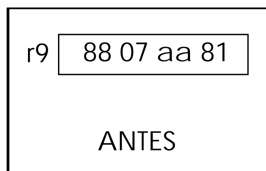


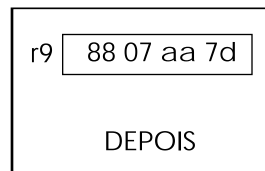


SUB	Subtração			
	Formato	Operação	Flags	Codificação
	sub *rdst*, *val8*	*rdst* ← *rdst* – *ext(imm8)*	OCSZ	0D | imm8 | rdst | –
	sub *rdst*, *rsrc*	*rdst* ← *rdst* – *rsrc*	OCSZ	0E | – | rdst | rsrc

Exemplo 4.14:

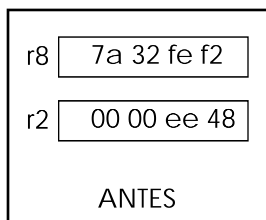
```
sub     r9, 4          ; instrução subtrai com valor imediato
```

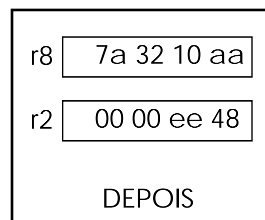




Exemplo 4.15:

```
sub     r8, r2         ; exemplo de instrução subtrai
```





Exemplo 4.16: Considere o seguinte comando de atribuição em Pascal:

```
var a, b, c: integer;
...
          a := b + c - 2;
...
```

Escreva um trecho de programa em linguagem de montagem que o implemente.

```
; trecho para implementar a := b + c - 2                    1
                                                            2
          ld       r0, b                                    3
          ld       r1, c                                    4
          add      r0, r1                                   5
          sub      r0, 2                                    6
          st       a, r0                                    7
...                                                         8
.data                                                       9
; reserva espaço para as variáveis inteiras a, b e c       10
a:        ds       4                                       11
b:        ds       4                                       12
c:        ds       4                                       13
```

4.3 Instruções de desvio

Até agora, em todos os exemplos mostrados as instruções são executadas em seqüência estrita das suas posições da memória, ou seja, o valor do registrador *ip* é sempre incrementado de quatro a cada instrução. Se não houver uma forma de fazer o valor de *ip* variar de maneira não seqüencial, fica impossível de implementar repetições como os comandos *for* e *while* em C ou Pascal. Para controlar o fluxo de execução do programa, são usadas instruções de desvio, que alteram o valor do registrador interno *ip*. Instruções de desvio podem ser condicionais ou incondicionais.

4.3.1 Desvio incondicional

A execução da instrução de desvio incondicional é bastante simples:

o valor do registrador *ip* é modificado para o valor do operando. Por exemplo, um laço infinito em linguagem de montagem é especificado assim:

```
iniloop:
        jmp iniloop
```

O exemplo acima mostra a sintaxe de um desvio incondicional com operando imediato. O Faíska possui também uma instrução de desvio incondicional onde o operando é um registrador:

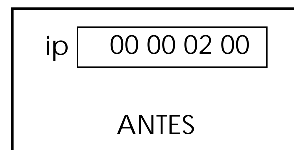
```
        jmp     r1
```

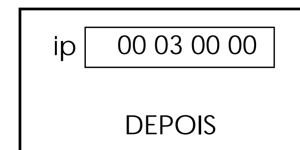
que copia o valor de *r1* no registrador *ip* (ou seja, desvia para o endereço "apontado" por *r1*).

JMP	Desvio incondicional			
	Formato	Operação	Flags	Codificação
	jmp *label*	*ip* ← *imm32*	–	0F | – | – | – imm32
	jmp *rdst*	*ip* ← *rdst*	–	10 | – | rdst | –

Exemplo 4.17:

```
        org   200h
        jmp   fora          ; exemplo de instrução de desvio incondicional
                            ; montada na posição 200h
                            ; codificação desta instrução utiliza duas palavras:
                            ; 0f 00 00 00    e    00 03 00 00
        ...
        org   30000h
fora:
        ...
```

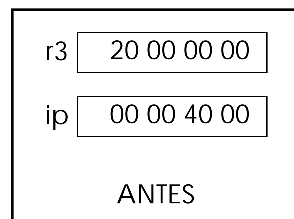


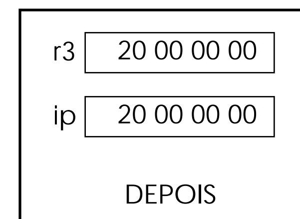


Lembre que o apontador de instruções *ip* não é um registrador de propósito geral, e não pode ser acessado diretamente pelo usuário. Ele é mostrado nas figuras apenas para ilustrar o funcionamento do processador.

Exemplo 4.18:

```
        org   400h
        jmp   r3            ; exemplo de instrução de desvio incondicional
                            ; montada na posição 400h
                            ; codificação desta instrução é 10 00 03 00
```





4.3.2 Desvios condicionais

Instruções de desvio condicional são mais interessantes. Elas executam ou não o desvio dependendo do estado de um conjunto de *bits de es-*

tado, que por sua vez são alterados pela execução de algumas instruções, em particular as instruções aritméticas. Os bits de estado guardam a informação do resultado da última operação aritmética ou lógica executada. Eles não são alterados por operações de transferência de dados, ou por instruções de desvio. No Faíska, os bits de estado são

- C: carry (vai-um). Ligado se operação causou vai-um ou vem-um, desligado caso contrário.
- Z: zero. Ligado se o resultado foi zero, desligado caso contrário.
- S: sinal. Cópia do bit mais significativo do resultado; considerando aritmética com sinal, se S igual a zero, o número é maior ou igual a zero. Se S igual a 1, resultado é negativo.
- O: *overflow*. Ligado se ocorreu estouro de campo; calculado como o ou-exclusivo entre o *carry-in* e o *carry-out* do bit mais significativo do resultado.

No descritor de instruções o campo *Flag* indica quais os bits de estado são afetados pela instrução.

Jcond	Desvia se *cond* é verdadeira			
	Formato	Operação	Flags	Codificação
	jcond label	se *cond* *ip* $\leftarrow$ *ip + ext(imm8)*	–	*codif* | imm8 | – | –

No Faíska, as operações de desvio condicional são

instr.	nome	cond	codif.
jc	desvia se carry	CF = 1	11
jnc	desvia se não carry	CF = 0	12
jz	desvia se zero	ZF = 1	13
jnz	desvia se diferente de zero	ZF = 0	14
jo	desvia se overflow	OF = 1	15
jno	desvia se não overflow	OF = 0	16

instr.	nome	cond	codif.
js	desvia se sinal igual a um	SF = 1	17
jns	desvia se sinal igual a zero	SF = 0	18
jl	desvia se menor (com sinal)	SF ≠ OF	19
jle	desvia se menor ou igual (com sinal)	SF ≠ OF ou ZF = 0	1A
jg	desvia se maior (com sinal)	SF = OF e ZF = 0	1B
jge	desvia se maior ou igual (com sinal)	SF = OF	1C
ja	desvia se “acima” (maior ,sem sinal)	CF = 0 e ZF = 0	1D

As instruções *jl* e *jg* (em inglês, respectivamente *jump if less* e *jump if greater*) devem ser usadas na comparação números inteiros com sinal; as instruções *jc*, *jnc* e *ja* (em inglês, *jump if above*) devem ser usadas na comparação de números inteiros sem sinal.

Se um desvio condicional não puder ser montado porque o endereço alvo ficou mais distante que o permitido, é necessário inverter a condição e utilizar o desvio incondicional. Por exemplo, se no trecho abaixo o rótulo *muito_longe* estiver muito distante do desvio condicional, de tal forma que a distância relativa não possa ser representada no campo *imm8* da instrução *jnc*,

```
        sub     r20, r21                                1
        jnc     muito_longe                             2
        add     r0, r1                                  3
        ...                                             4
muito_longe:                                            5
```

podemos re-escrever o trecho utilizando um desvio incondicional:

```
        sub     r20, r21                                1
        jc      bem_perto       ; invertemos a condição 2
        jmp     muito_longe                             3
bem_perto:                                              4
        add     r0, r1                                  5
        ...                                             6
muito_longe:                                            7
```

Exemplo 4.19: 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:                                                         1
        mov   r0, r1        ; vamos usar r0 como rascunho       2
        sub   r0, r2        ; r1 maior que r2?                  3
        ja    final         ; sim --- nada a fazer              4
        mov   r0, r1        ; troca r1 com r2...                5
        mov   r1, r2        ; usando r0...                      6
        mov   r2, r0        ; como temporário                   7
final:                                                          8
        ...                                                     9
```

Exemplo 4.20: 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.

```
maior:                                                                  1
        mov   r4, r1        ; usando r4 como rascunho                   2
        sub   r4, r2        ; compara r1 com r2                         3
        jge   um            ; desvia se r1 maior ou igual a r2          4
        mov   r4, r1        ; guarda maior valor entre (r1,r2) em r4... 5
        mov   r0, r1        ; e em r0                                   6
        jmp   outro                                                     7
um:                                                                     8
        mov   r4, r2        ; guarda maior valor entre (r1,r2)em r4...  9
        mov   r0, r2        ; e em r0                                  10
outro:                                                                 11
        sub   r4, r3        ; compara maior entre (r1, r2) com r3      12
        jg    final         ; r3 é menor, r0 já tem maior valor        13
        mov   r0, r3        ; maior é r3                               14
final:                                                                 15
        ...                                                            16
```

4.3.3 Nova instrução aritmética: cmp

O exemplo acima, que deveria ser bem simples, ficou um pouco mais extenso e complicado porque a instrução de subtração usada para comparar os registradores altera o valor do registrador destino. Isto faz com que precisemos usar registradores auxiliares que devem ser reinicializados após cada subtração. Como comparação é uma operação relativamente freqüente, ela é definida como uma instrução primitiva na

maioria dos processadores. Esta nova instrução é bastante semelhante à instrução *sub*, com a diferença de que o registrador destino não é alterado. Quer dizer, a subtração é efetuada, os bits de estado são modificados correspondentemente, mas o resultado não é colocado no registrador destino, que permanece com o valor inicial.

CMP	Comparação						
	Formato	Operação	Flags	Codificação			
	sub *rdst, imm8*	*rdst – ext(imm8)*	OCSZ	1E	imm8	rdst	–
	sub *rdst, rsrc*	*rdst – rsrc*	OCSZ	1F	–	rdst	rsrc

Vamos re-escrever a solução para o Exemplo 4.20 com a instrução *cmp*.

```
maior:                                                          1
        mov     r0, r1       ; assume que r1 é o maior          2
        cmp     r1, r2       ; compara r1 e r2                  3
        jg      outro        ; desvia se r1 > r2                4
        mov     r0, r2       ; r2 é maior (ou igual!), guarda em r0   5
outro:                                                          6
        cmp     r0, r3       ; compara maior entre r1 e r2 com r3     7
        jg      final        ; r0 já tem o maior valor          8
        mov     r0, r3       ; maior é r3                       9
final:                                                          10
        ...                  ; final do trecho                  11
```

Exemplo 4.21: Escreva um trecho de programa que determine qual o maior valor de um vetor de números de 32 bits, sem sinal, apontado por *r2*. Inicialmente, *r3* contém o número de valores presentes no

vetor; assuma *r3* > 0. Ao final do trecho, *r0* deve conter o valor máximo e *r1* sua posição.

```
; primeira versão...                                                          1
; trecho que determina maior valor de vetor de números sem sinal (32 bits)    2
; entrada:                                                                    3
;        r2: endereço do primeiro elemento do vetor                           4
;        r3: número de elementos do vetor                                     5
; saída:                                                                      6
;        r0: maior elemento                                                   7
;        r1: posição do maior elemento                                        8
inicio:                                                                       9
        ld      r1, r2        ; guarda apontador                              10
        mov     r0, (r1)      ; e valor maximo                                11
proximo:                                                                      12
        add     r2, 4         ; avanca ponteiro                               13
        sub     r3, 1         ; conta este elemento                           14
        jz      final         ; terminou de verificar toda a cadeia?          15
        ld      r4, (r2)      ; não, então toma mais um valor                 16
        cmp     r0, r4        ; compara com maior corrente                    17
        jnc     proximo       ; volta se menor que o que conhecemos           18
        jmp     inicio        ; achamos um maior, guarda novo...              19
                              ; apontador e novo máximo                       20
final:                                                                        21
        ...                   ; final do trecho                               22
```

Note que o registrador que percorre a cadeia, *r2*, é incrementado de quatro a cada elemento verificado, pois cada elemento ocupa quatro bytes.

Exemplo 4.22: Escreva um trecho de programa para mover um bloco de palavras de memória da posição apontada por *r1* para a posição apontada por *r2*. O tamanho do bloco (número de palavras, assuma diferente de zero) que deve ser movido é dado em *r3*.

```
; primeira versão...                                                          1
; trecho que move bloco de palavras                                           2
; entrada:                                                                    3
;        r1: endereço inicial do bloco                                        4
;        r2: endereço para onde o bloco deve ser movido                       5
;        r3: tamanho do bloco, em palavras de 32 bits, diferente de zero      6
MoveBloco:                                                                    7
        ld      r0, (r1)      ; copia palavra                                 8
        st      (r2), r0      ; na nova posição                               9
        add     r1, 4         ; avança ponteiros                              10
```

```
add     r2, 4                                                  11
sub     r3, 1          ; verifica se chegou ao final do bloco  12
jnz     MoveBloco      ; não, continua movendo                 13
...                    ; aqui quando terminou de mover o bloco 14
```

Infelizmente esta solução simples não funciona para todos os casos. A Figura 4.2 ilustra de forma esquemática as configurações possíveis para este problema, dependendo do tamanho do bloco e de suas posições inicial e final.

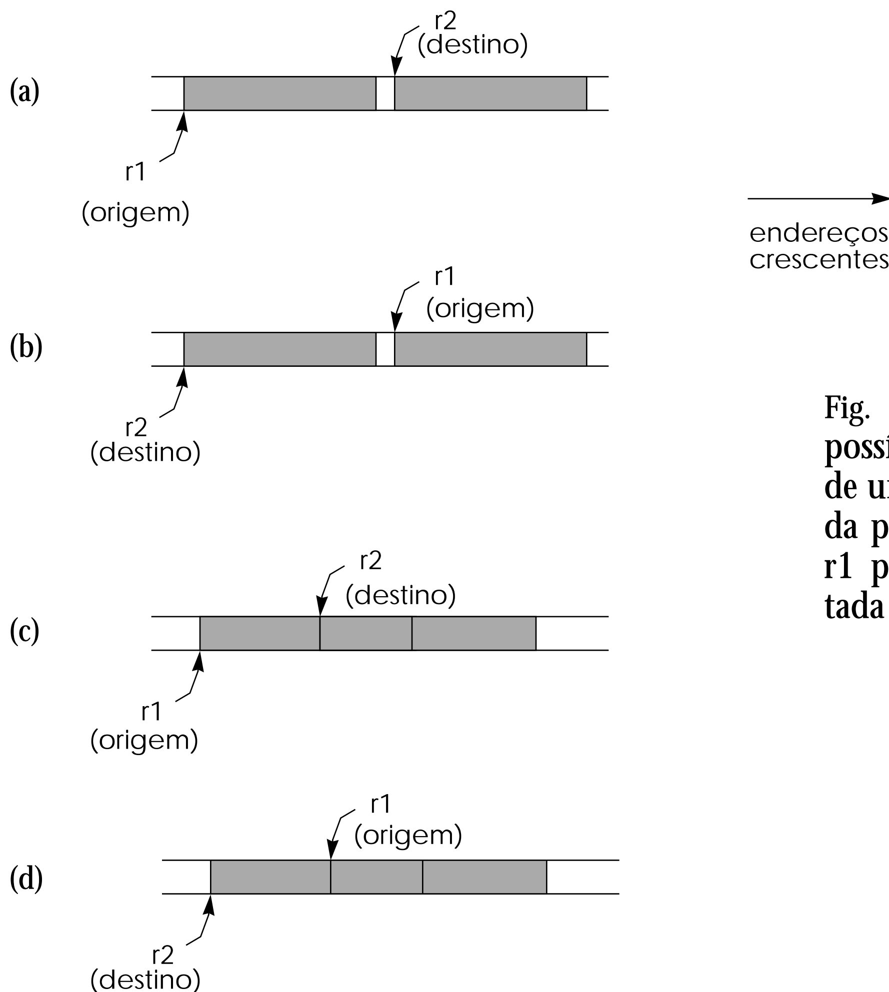


Fig. 4.2 Configurações possíveis no movimento de um bloco de memória da posição apontada por r1 para a posição apontada por r2.

A solução apresentada funciona perfeitamente para os casos (a) e (b) da Figura 4.2, quando não há interseção entre a posição original e a posição final do bloco, e funciona também para o caso (d).

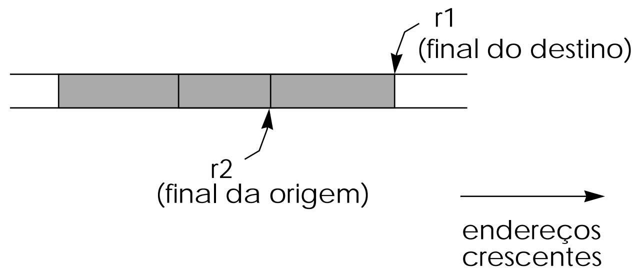


Fig. 4.3 Deslocando os apontadores para o final das posições original e destino dos blocos.

Mas no caso (c) a cópia das primeiras palavras destrói o conteúdo do bloco original. Neste caso, é necessário fazer a cópia do final do bloco para o começo, como esquematizado na Figura 4.3.

```
; segunda versão...                                                          1
; move bloco de palavras                                                     2
; entrada:                                                                   3
;       r1: endereço inicial do bloco                                        4
;       r2: endereço para onde o bloco deve ser movido                       5
;       r3: tamanho do bloco, em palavras de 32 bits, diferente de zero      6
MoveBloco:                                                                   7
        set     r5, 4         ; vamos usar r5 para avançar os ponteiros      8
        cmp     r1, r2        ; inicialmente verifica qual caso              9
        jz      final         ; origem = destino?? nada a fazer              10
        jc      MvBloco       ; casos (b) e (d), origem é menor...           11
                              ; que destino, move do inicio para o fim       12
; aqui tratamos casos (a) e (c) -- embora apenas (c) cause problema          13
; é mais fácil tratá-los juntos; move do fim para o início                   14
        set     r5, -4        ; o "avanço" vai ser para trás                 15
        mov     r0, r3        ; vamos calcular o final do bloco, r0 é rascunho 16
        add     r0, r0        ; r0 = 2*comprimento                           17
        add     r0, r0        ; r0 = 4*comprimento, ou seja, número de bytes 18
        sub     r0, 4                                                        19
        add     r1, r0        ; r1 aponta para última palavra da origem      20
        add     r2, r0        ; r2 aponta para última palavra da destino     21
MvBloco:                                                                     22
        ld      r0, (r1)      ; copia palavra                                23
        st      (r2), r0      ; na nova posição                              24
        add     r1, r5        ; avança (ou retrocede) ponteiros              25
        add     r2, r5                                                       26
        sub     r3, 1         ; verifica se chegou ao final do bloco         27
        jnz     MvBloco       ; não, continua movendo                        28
final:                                                                       29
        ...                   ; aqui quando terminou de mover o bloco        30
```

Exemplo 4.23: Escreva um trecho de programa que verifique se uma cadeia de bytes é palíndrome (ou seja, dá o mesmo resultado quando lida da direita para a esquerda ou da esquerda para a direita; “miauaim” é um exemplo). O endereço do início da cadeia é dado em *r2*, e o seu comprimento (maior ou igual a 0) é dado em *r1*. Se a cadeia é palíndrome, o registrador *r0* deve valer zero ao final; caso contrário, *r0* deve ter o valor 1 ao final.

```
;trecho para verificar se cadeia de bytes é palíndrome                          1
;       início: endereço inicial da cadeia em r2 e seu comprimento em r1        2
;       final: r0 igual a 0 se palíndrome, 1 caso contrário                     3
                                                                                4
Palíndrome:                                                                     5
        cmp   r1, 0       ; se cadeia vazia, palavra é palíndrome               6
        jz    palind      ; vai retornar com r0 igual a 0                       7
        mov   r3, r2      ; vamos fazer ...                                     8
        add   r3, r1      ; r3 apontar ...                                      9
        sub   r3, 1       ; para último elemento da cadeia                      10
proxbyte:                                                                       11
        ldb   r0, (r2)    ; elemento do início                                  12
        ldb   r1, (r3)    ; elemento do final                                   13
        cmp   r0, r1      ; campara os elementos                                14
        jnz   naopal      ; se diferentes, palavra não é palíndrome             15
        add   r2, 1       ; avança ponteiro do inicio                           16
        sub   r3, 1       ; retrocede ponteiro do final                         17
        cmp   r2, r3      ; verifica se cadeia terminou                         18
        jnc   palind      ; sim, terminou, a palavra é palíndrome               19
        jmp   proxbyte    ; continua verificação                                20
naopal:                                                                         21
        set   r0, 1                                                             22
        jmp   final       ; termina com r0 igual a 1                            23
palind:                                                                         24
        set   r0, 0       ; termina com r0 igual a 0                            25
final:                                                                          26
```

Esta solução pode ser melhorada. Primeiramente, note que existe uma seqüência de dois desvios, o primeiro condicional e o segundo incondicional, nas linhas linhas 19~22. Quando isto ocorre, podemos em geral rearrumar o código de maneira a utilizar apenas um desvio. Uma outra modificação possível é a eliminação do teste para verificar se a cadeia é vazia no início (linhas 6~9), incorporando este caso no tratamento geral. A nova versão fica assim:

```
; Palíndrome - segunda versão                                                     1

;trecho para verificar se cadeia de bytes é palíndrome                           1
;        início: endereço inicial da cadeia em r2 e seu comprimento em r1        2
;        final: r0 igual a 0 se palíndrome, 1 caso contrário                     3
;        destroi: r1, r2 e r3                                                    4
Palíndrome:                                                                      5
        cmp    r1, 0          ; se cadeia vazia, palavra é palíndrome            6
        jz     palind         ; vai retornar com r0 igual a 0                    7
        mov    r3, r2         ; vamos fazer ...                                  8
        add    r3, r1         ; r3 apontar ...                                   9
        sub    r3, 1          ; para último elemento da cadeia                   10
proxbyte:                                                                        11
        cmp    r2, r3         ; verifica se cadeia terminou                      12
        jnc    palind         ; sim, terminou, a palavra é palíndrome            13
        ldb    r0, (r2)       ; elemento do inicio                               14
        ldb    r1, (r3)       ; elemento do final                                15
        add    r2, 1          ; avança ponteiro do inicio                        16
        sub    r3, 1          ; retrocede ponteiro do final                      17
        cmp    r0, r1         ; campara os elementos                             18
        jz     proxbyte                                                          19
        set    r0, 1                                                             20
        jmp    final          ; termina com r0 igual a 1                         21
palind:                                                                          22
        set    r0, 0          ; termina com r0 igual a 0                         23
final:                                                                           24
```

4.3.4 Tabelas de desvios

Uma técnica bastante útil em linguagem de montagem é a utilização de tabelas de desvio. Para ilustrar o seu uso, vamos examinar como pode ser implementado um comando de seleção como *switch* em C ou *case* em Pascal.

Considere o seguinte trecho de programa em C:

```
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;
}
```

Uma maneira de implementar este trecho em linguagem de montagem é comparar o valor de *val* seqüencialmente com cada um das possíveis seleções, como no trecho em linguagem de montagem abaixo:

```
; comando switch (val)                                          1
        ld      r0, val         ; carrega variavel de seleção   2
        set     r1, 1000        ; primeira selação              3
        cmp     r0, r1          ; verifica se igual             4
        jnz     sel2            ; desvia se diferente           5
; case 1000                                                     6
        ld      r0, y                                           7
        st      x, r0           ; x = y                         8
        jmp     final           ; break                         9
sel2:                                                           10
        set     r1, 1001        ; segunda selação               11
        cmp     r0, r1          ; verifica se igual             12
        jnz     sel3            ; desvia se diferente           13
; case 1001                                                     14
        ld      r0, x                                           15
        st      y, r0           ; y = x                         16
        jmp     final           ; break                         17
sel3:                                                           18
        set     r1, 1004        ; terceira selação              19
        cmp     r0, r1          ; verifica se igual             20
        jnz     sel4            ; desvia se diferente           21
; case 1004                                                     22
        ld      r0, x                                           23
        st      t, r0           ; t = x                         24
        jmp     s1005           ; note que não há break         25
sel4:                                                           26
        set     r1, 1005        ; quarta selação                27
        cmp     r0, r1          ; verifica se igual             28
```

```
        jnz     sel5            ; desvia se diferente                      29
; case 1005                                                                30
s1005:                                                                     31
        set     r0, 0                                                      32
        st      t, r0           ; t = 0                                    33
        jmp     final           ; break                                    34
; default                                                                  35
sel5:                                                                      36
        set     r0, 0                                                      37
        st      t, r0           ; t = 0                                    38
        st      x, r0           ; x = 0                                    39
        st      y, r0           ; y = 0                                    40
final:                                                                     41
        ...                                                                42
```

Esta solução é aplicável quando o número N de seleções é pequeno. Mas note que a cada execução do comando switch são realizadas em média $N/2$ comparações. Se N é grande, a execução do comando será (em média) demorada. Nesse caso o mais eficiente é usar uma tabela de desvios, que é simplesmente uma tabela onde cada elemento é um rótulo:

```
tabela_desvio:
                dw      ponto1
                dw      ponto2
                dw      ponto3
                ...
                dw      pontoN
```

Para efetuar o desvio para a posição referente ao segundo elemento da tabela, basta carregar este elemento em um registrador e utilizar a instrução de desvio por registrador.

Uma solução para o comando *switch* acima utilizando uma tabela de desvio é:

```
; comando switch (val)                                                     1
        ld      r0, val         ; carrega variavel de seleção              2
        set     r1, 1000        ; primeiro verificamos os limites          3
        cmp     r0, r1          ; menor que menor entrada na tabela?       4
        jc      casedefault     ; sim, desvia                              5
        set     r1, 1005        ; compara com maior valor                  6
        cmp     r1, r0                                                     7
        jc      casedefault     ; val é maior que a maior entrada na tabela 8
        set     r1, tab_switch  ; carrega endereço da tabela de desvios    9
                                ; r0 será o índice na tabela               10
```

```
        sub     r0, 1000       ; primeiro valor tem índice zero                 11
        add     r0, r0         ; cada elemento na tabela tem 4 bytes...         12
        add     r0, r0         ; portanto multiplicamos índice por 4            13
        add     r0, r1         ; r0 aponta para elemento                        14
        ld      r0, (r0)       ; pega elemento na tabela                        15
        jmp     r0             ; e desvia para seleção correspondente           16
tab_switch:                                                                     17
        dw      case1000                                                        18
        dw      case1001                                                        19
        dw      casedefault                                                     20
        dw      casedefault                                                     21
        dw      case1004                                                        22
        dw      case1005                                                        23
case1000:                                                                       24
        ld      r0, y                                                           25
        st      x, r0          ; x = y                                          26
        jmp     final          ; break                                          27
case1001:                                                                       28
        ld      r0, x                                                           29
        st      y, r0          ; y = x                                          30
        jmp     final          ; break                                          31
case1004:                                                                       32
        ld      r0, x                                                           33
        st      t, r0          ; t = x                                          34
                               ; note que não há break                          35
case1005:                                                                       36
        set     r0, 0                                                           37
        st      t, r0          ; t = 0                                          38
        jmp     final          ; break                                          39
casedefault:                                                                    40
        set     r0, 0                                                           41
        st      t, r0          ; t = 0                                          42
        st      x, r0          ; x = 0                                          43
        st      y, r0          ; y = 0                                          44
final:                                                                          45
        ...                                                                     46
```

Note que nesta abordagem o número de comparações a cada execução do comando switch é fixo.

4.4 Instruções Lógicas

No grupo das instruções lógicas, como o nome indica, estão as instruções que efetuam operações lógicas como *e*, *ou* e *ou-exclusivo.* No Faíska, as instruções lógicas operam somente sobre registradores. Outros

processadores, como os da família Intel x86, admitem que um dos operandos esteja na memória.

As operações lógicas para operandos de apenas um bit são descritas na tabela Tabela 4.1. As instruções lógicas que implementam as operações *e*, *ou* e *ou-exclusivo* operam bit a bit, isto é, bit 0 do primeiro operando com bit 0 do segundo operando, bit 1 com bit 1, e assim por diante.

Tabela 4.1 Operações lógicas com operandos de 1 bit.

op1	op2	e	ou	ou-exclusivo
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

AND	E lógico			
	Formato	Operação	Flags	Codificação
	and rdst, rsrc	*rdst* ← *rdst* and *rsrc*	OCSZ	20 | – | rdst | rsrc

Exemplo 4.24:

```
and r2, r3                ; exemplo de instrução e_lógico
```

ANTES		DEPOIS	
r3	77 ea 12 66	r3	77 ea 12 66
r2	48 00 e8 f2	r2	40 00 00 62

OR	OU lógico			
	Formato	Operação	Flags	Codificação
	or rdst, rsrc	*rdst* ← *rdst* or *rsrc*	OCSZ	21 | – | rdst | rsrc

Exemplo 4.25:

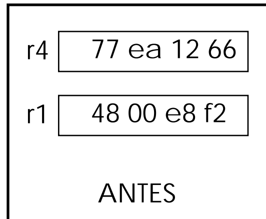
```
or      r0, r4          ; exemplo de instrução ou_lógico
```

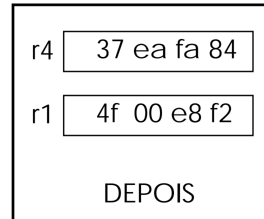
	ANTES	DEPOIS
r4	77 ea 12 66	77 ea 12 66
r0	48 00 e8 f2	7f ea fa f6

XOR	OU EXCLUSIVO lógico			
	Formato	Operação	Flags	Codificação
	xor rdst, rsrc	*rdst* ← *rdst* xor *rsrc*	OCSZ	22 | – | rdst | rsrc

Exemplo 4.26:

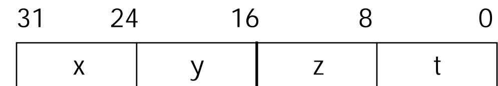
```
xor     r4, r1                ; exemplo de instrução ou_exclusivo
```





A instrução *and* pode ser usada por exemplo para isolar alguns bits de uma palavra de memória. Suponha que uma palavra de 32 bits está sendo usada para armazenar 8 elementos de uma estrutura, onde cada elemento é um número de 4 bits, como na declaração Pascal

```
var
    a: register
        x, y, z, t: packed [0..255];
    end;
```



Para isolar apenas o elemento *t*, podemos utilizar

```
mov    r10, a        ; carrega estrutura
ld     r0, 0ffh      ; máscara 00 00 00 ff
and    r10, r0       ; r10 tem o elemento t isolado
```

Para isolar *z*, o segundo elemento da estrutura, podemos nos valer da mesma idéia, mas para obter o valor correto do elemento nesse caso é necessário primeiro deslocar a estrutura de 8 bits para a direita, antes de isolar o elemento:

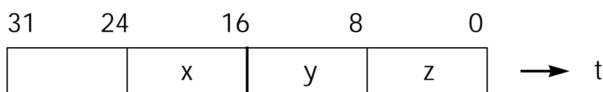


Fig. 4.4 Deslocando a estrutura de 8 bits para a direita

Os processadores incluem no repertório instruções que permitem di-

ferentes formas de efeturar deslocamentos deste tipo: para a direita, para a esquerda, utilizando o bit de estado C, e outros. Estas instruções são detalhadas a seguir.

4.4.1 Instruções de deslocamento e rotação

O Faíska possui as seguintes instruções de deslocamento e rotação:

shl	deslocamento para a esquerda
shr	deslocamento para a direita
sar	deslocamento aritmético para a direita
rol	rotação para a esquerda
ror	rotação para a direita
rcl	rotação para a esquerda com o bit de estado C
rcr	rotação para a direita com o bit de estado C

A Figura 4.5 resume de forma esquemática as instruções de deslocamento no Faíska.

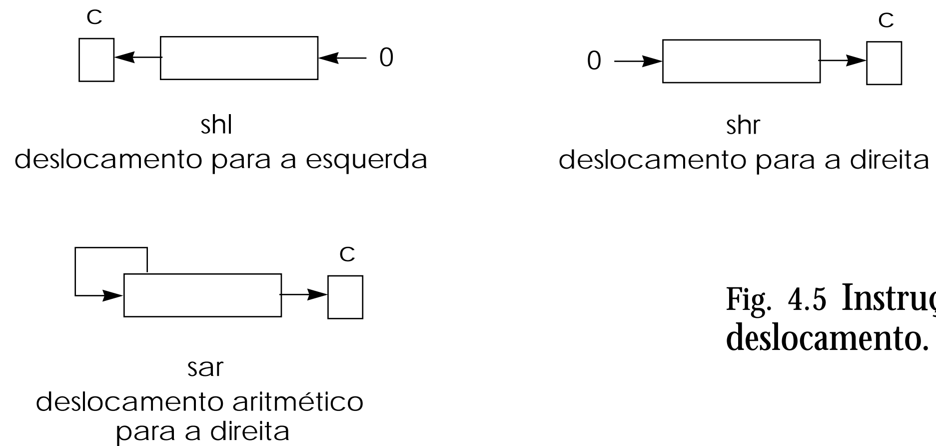


Fig. 4.5 Instruções de deslocamento.

SHR	Deslocamento lógico para a direita						
	Formato	Operação	Flags	Codificação			
	shr dst, val5	repeat *imm5* times { $rdst_i \leftarrow rdst_{i+1}$ C $\leftarrow rdst_0, rdst_{31} \leftarrow 0$ }	C	23	imm5	rdst	–
	shr dst, rsrc	repeat *rsrc* times { $rdst_i \leftarrow rdst_{i+1};$ C $\leftarrow rdst_0 ; rdst_{31} \leftarrow 0$ }	C	24	–	rdst	rsrc

SHL	Deslocamento lógico para a esquerda						
	Formato	Operação	Flags	Codificação			
	shr dst, val5	repeat *imm5* times { $rdst_{i+1} \leftarrow rdst_i$ C $\leftarrow rdst_{31}\ rdst_0 \leftarrow 0$ }	C	25	imm5	rdst	–
	shr dst, rsrc	repeat *rsrc* times { $rdst_{i+1} \leftarrow rdst_i$ C $\leftarrow rdst_{31} ; rdst_0 \leftarrow 0$ }	C	26	–	rdst	rsrc

Operações de deslocamento são muito úteis para efetuar a divisão ou multiplicação de um número inteiro por uma potência de 2. Como visto no Capítulo 2, cada deslocamento de um bit para a direita divide o operando por dois, e cada deslocamento de um bit para a esquerda multiplica o operando por 2, se o operando é um número inteiro positivo. Se o operando é um número inteiro negativo, a instrução de deslocamento lógico para a direita não pode ser usada para dividí-lo por 2, pois como zeros são injetados o resultado será sempre positivo. Por esta razão uma outra instrução de deslocamento é fornecida pelo Faíska:

sar   deslocamento aritmético para a direita

A diferença entre o deslocamento lógico e o aritmético é que no arit-

mético ao invés de serem injetados zeros, o bit mais significativo do operando é usado na injeção. Ou seja, o bit de sinal do operando é repetido e deslocado para a direita. Assim, quando o operando representar um número inteiro com sinal em complemento de dois, se o valor inicial é negativo, o resultado também o será (e valerá metade do valor inicial). Observe que a instrução simétrica *sal* não é necessária.

SAR	Deslocamento aritmético para a direita			
	Formato	Operação	Flags	Codificação
	sar *rdst*, *imm5*	repeat *imm5* times { $rdst_i \leftarrow rdst_{i+1}$ C ← $rdst_0$, $rdst_{31} \leftarrow 0$ }	C	27 | imm5 | rdst | –
	sar *rdst*, *rsrc*	repeat *rsrc* times { $rdst_i \leftarrow rdst_{i+1}$; C ← $rdst_0$; $rdst_{31} \leftarrow 0$ }	C	28 | – | rdst | rsrc

Instruções de rotação

As instruções de rotação são bastante similares às instruções de deslocamento, mas ao invés de novos bits serem injetados, os bits do próprio operando são re-injetados a cada bit deslocado. A Figura 4.6 ilustra de forma esquemática as instruções de rotação disponíveis no Faíska.

Note que na rotação para a esquerda o bit de sinal é copiado também no bit de estado *C*, além de ser re-injetado no bit menos significativo. A operação simétrica acontece na rotação para a direita.

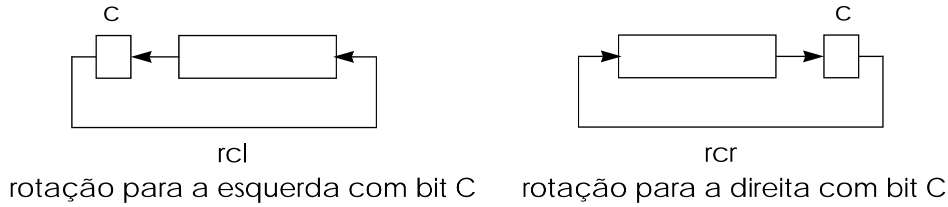


Fig. 4.6 **Instruções de rotação**

ROL	Rotação para a esquerda			
	Formato	Operação	Flags	Codificação
	rol *rdst*, *imm5*	repeat *imm5* times { $rdst_{i+1} \leftarrow rdst_i$ C, $rdst_0 \leftarrow rdst_{31}$ }	C	29 | imm5 | rdst | –
	rol *rdst*, *rsrc*	repeat *rsrc* times { $rdst_{i+1} \leftarrow rdst_i$ C, $rdst_0 \leftarrow rdst_{31}$ }	C	2A | – | rdst | rsrc

ROR	Rotação para a direita			
	Formato	**Operação**	**Flags**	**Codificação**
	ror *rdst*, *imm5*	repeat *imm5* times { $rdst_i \leftarrow rdst_{i+1}$; C, $rdst_{31} \leftarrow rdst_0$ }	C	2B | imm5 | rdst | –
	ror *rdst*, *rsrc*	repeat *rsrc* times { $rdst_i \leftarrow rdst_{i+1}$; C, $rdst_{31} \leftarrow rdst_0$ }	C	2C | – | rdst | rsrc

RCL	Rotação com carry para a esquerda			
	Formato	**Operação**	**Flags**	**Codificação**
	rol *rdst*, *imm5*	repeat *imm5* times { $rdst_{i+1} \leftarrow rdst_i$ $rdst_0 \leftarrow$ C; C $\leftarrow rdst_{31}$ }	C	2D | imm5 | rdst | –
	rol *rdst*, *rsrc*	repeat *rsrc* times { $rdst_{i+1} \leftarrow rdst_i$ $rdst_0 \leftarrow$ C; C $\leftarrow rdst_{31}$ }	C	2E | – | rdst | rsrc

RCR	Rotação para a direita			
	Formato	Operação	Flags	Codificação
	rcr *rdst*, *imm5*	repeat *imm5* times { $rdst_i \leftarrow rdst_{i+1}$ C $\leftarrow rdst_0$; $rdst_{31} \leftarrow$ C }	C	2F | imm5 | rdst | –
	rcr *rdst*, *rsrc*	repeat *rsrc* times { $rdst_i \leftarrow rdst_{i+1}$ C $\leftarrow rdst_0$; $rdst_{31} \leftarrow$ C }	C	30 | – | rdst | rsrc

Exemplo 4.27: Escreva um trecho de programa que troque os bits mais e menos significativos de *r0*, sem alterar os bits restantes.

Antes

Depois

```
; trecho para inverter o bit mais significativo com o menos              1
; significativo de r0                                                     2
        xor     r1, r1          ; zera r1, vamos montar y00..00x em r1    3
        shl     r0, 1           ; C ← x                                   4
        rcl     r1, 2           ; r1 tem 00...0x0                         5
        shr     r0, 2           ; C ← y, r0 tem 00bb...b, onde b é digito 6
        rcr     r1, 2           ; r1 tem y00...00x                        7
        shl     r0, 1           ; r0 tem 0bb...bb0                        8
        or      r0, r1          ; e bits são invertidos!                  9
```

As instruções do grupo lógico podem ser utilizadas para implementar eficientemente o tipo *conjunto* de linguagens como Pascal. Suponha que um conjunto de cardinalidade máxima 32 deva ser implementado. O conjunto pode ser representado no Faíska por uma palavra de memória onde cada bit corresponde a um elemento, usando a convenção de que o valor 1 de um bit representa a presença do elemento correspondente do conjunto, e o valor 0 a ausência.

Para incluir o *i*-ésimo elemento em um conjunto *a_set*, onde *i* representa um número de 1 a 32 armazenado em r1, podemos usar o seguinte trecho de código

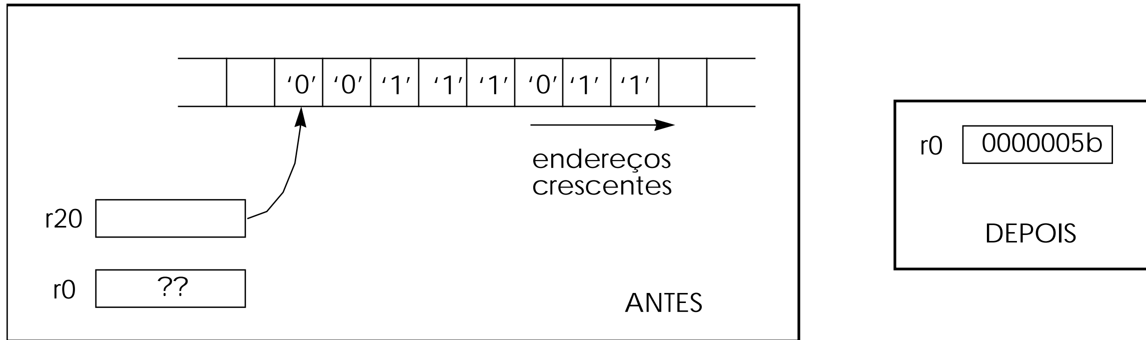
```
; na entrada, r1 tem valor entre 1 e 32                          1
        ld      r2, a_set       ; um conjunto qualquer           2
        sub     r1,1            ; r1 agora entre 0 e 31          3
        set     r0, 1           ; vamos usar como máscara        4
        shl     r0, r1          ; desloca para a posição         5
                                ; correspondente ao elemento     6
        or      r0, r2          ; inclui elemento                7
```

Para verificar se o i-ésimo elemento está presente, o código é similar:

```
; na entrada, r1 tem valor entre 1 e 32                              1
        ld      r2, a_set       ; um conjunto qualquer               2
        sub     r1,1            ; r1 agora entre 0 e 31              3
        set     r0, 1           ; vamos usar como máscara            4
        shl     r0, r1          ; desloca para a posição             5
                                ; correspondente ao elemento         6
        and     r0, r2          ; verifica se elemento está presente 7
        jz      not_present     ; desvia se não está presente        8
here_present:                                                        9
; aqui se o elemento está presente                                   10
        ...                                                          11
not_present:                                                         12
; aqui se o elemento não está presente                               13
        ...                                                          14
```

Exemplo 4.28: Suponha que *r20* aponta para uma cadeia de 8 caracteres '0' ou '1' que representa um número em notação binária (*r20* aponta para o "bit" mais significativo, como indica a figura abaixo). Escreva um trecho de programa que coloque em *r0* o valor que a cadeia representa.

```
; trecho para calcular valor binário representado por uma                1
; cadeia de caracteres '0' e '1' apontada por r20                         2
; primeira versão                                                         3
        set     r1, 8           ; vamos usar como contador                4
        xor     r0, r0          ; inicializa r0, vamos montar valor bit a bit 5
prox_dig:                                                                 6
        ld      r2, (r20)       ; r2 tem dígito: 30h ou 31h               7
        sub     r2, '0'         ; agora r2 tem o valor do dígito: 0 ou 1  8
        shl     r0, 1           ; prepara espaço para novo bit            9
```




```
        cmp   r2, 0                                                   10
        jz    continua     ; nada a fazer                             11
        or    r0, 1        ; monta novo bit com valor 1               12
continua:                                                             13
        add   r20, 1       ; avança apontador                         14
        sub   r1, 1        ; chegou ao final da cadeia?               15
        jnz   prox_dig     ; não, trata mais dígitos                  16
        ...                ; sim, término do trecho, valor do byte em r0   17
```

Podemos melhorar um pouco esta solução, eliminando a necessidade da comparação e teste das linhas 10~11:

```
; trecho para calcular valor binário representado por uma             1
; cadeia de caracteres '0' e '1' apontada por r20                      2
; segunda versão                                                      3
        set   r1, 8        ; vamos usar como contador                 4
        xor   r0, r0       ; inicializa r0, vamos montar valor bit a bit   5
prox_dig:                                                             6
        ld    r2, (r20)    ; r2 tem dígito: 30h ou 31h                7
        sub   r2, '0'      ; agora r2 tem o valor do dígito: 0 ou 1   8
        shl   r0, 1        ; prepara espaço para novo bit             9
        or    r0, r2       ; monta novo bit com valor 0 ou 1          10
continua:                                                             11
        add   r20, 1       ; avança apontador                         12
        sub   r1, 1        ; chegou ao final da cadeia?               13
        jnz   prox_dig     ; não, trata mais dígitos                  14
        ...                ; sim, término do trecho, valor do byte em r0   15
```

Ainda uma outra versão, desta vez usando operações de rotação com o bit de estado *C* para diminuir ainda mais o número de instruções do loop nas linhas 7~8:

```
; trecho para calcular valor binário representado por uma             1
; cadeia de caracteres '0' e '1' apontada por r20                      2
; terceira versão                                                     3
        set   r1, 8        ; vamos usar como contador                 4
```

```
        xor     r0, r0          ; inicializa r0, vamos montar valor bit a bit    5
prox_dig:                                                                        6
        ld      r2, (r20)       ; r2 tem dígito: 30h ou 31h                      7
        rcr     r2, r1          ; coloca novo bit 0 ou 1 no Carry                8
        rcl     r0, 1           ; monta novo bit                                 9
continua:                                                                        10
        add     r20, 1          ; avança apontador                               11
        sub     r1, 1           ; chegou ao final da cadeia?                     12
        jnz     prox_dig        ; não, trata mais dígitos                        13
        ...                     ; sim, término do trecho, valor do byte em r0    14
```

4.5 Mais instruções aritméticas

Em algumas aplicações é necessário representar e manipular números inteiros de mais de 32 bits. Para facilitar a adição e subtração de inteiros de mais de 32 bits o Faíska fornece duas operações aritméticas especiais, que levam em conta o valor corrente do bit de estado C:

ADC	Soma com bit de estado C			
	Formato	Operação	Flags	Codificação
	adc *rsrc, rdst*	*rdst* ← *rdst* + *rsrc* + *C*	OCSZ	| | – | rdst | rsrc |

SBC	Subtrai com bit de estado C			
	Formato	Operação	Flags	Codificação
	sbc *rsrc, rdst*	*rdst* ← *rdst* – *rsrc* – *C*	OCSZ	| | – | rdst | rsrc |

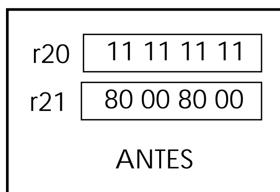
Por exemplo, para efetuar a soma de inteiros de 64 bits no Faíska é necessário realizar a operação em duas etapas. A primeira etapa efetua a adição dos 32 bits menos significativos, e a segunda efetua a adição dos 32 bits mais significativos. A adição das parcelas mais significativas dos

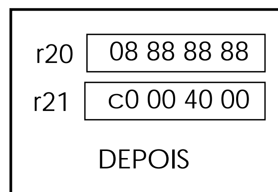
operandos deve levar em conta o vai-um da adição das parcelas menos significativas:

```
; trecho para calcular a = a+ b, onde a e b são variáveis inteiras de 64 bits   1
        ld      r0, a + 4      ; parcela menos significativa do operando a     2
        ld      r1, b + 4      ; parcela menos significativa do operando b     3
        add     r0, r1                                                          4
        st      a + 4, r0      ; guarda parcela menos significativa do         5
                               ; resultado                                      6
        ld      r0, a          ; parcela mais significativa do operando a      7
        ld      r1, b          ; parcela mais significativa do operando b      8
        adc     r0, r1         ; adiciona levando em conta vai-um              9
                               ; anterior                                      10
        st      a, r0          ; guarda parcela mais significativa do          11
                               ; resultado                                      12
```

Operações de rotação com o bit de estado *C* também podem ser utilizadas para manipular elementos de mais de 32 bits. Por exemplo, para dividir um número inteiro de 64 bits armazenado em *r20* (parcela mais significativa) e *r21* (parcela menos significativa), podemos fazer

```
        shr     r20, 1         ; flag C tem bit menos significativo da
                               ; parcela mais significativa
        rcr     r21, 1         ; completa deslocamento do elemento
```





4.6 Exercícios

1. Escreva um trecho de programa que troque os bits mais e menos significativos de *r0* e *r1*, sem alterar os bits restantes.

2. Descreva uma maneira de implementar conjuntos de mais de 32 elementos no Faíska.

3. Escreva um trecho de programa que verifique se os 32 bits do registrador R1 formam um padrão palíndrome (por exemplo, 80000001h e 55aa55aah são palíndromes).