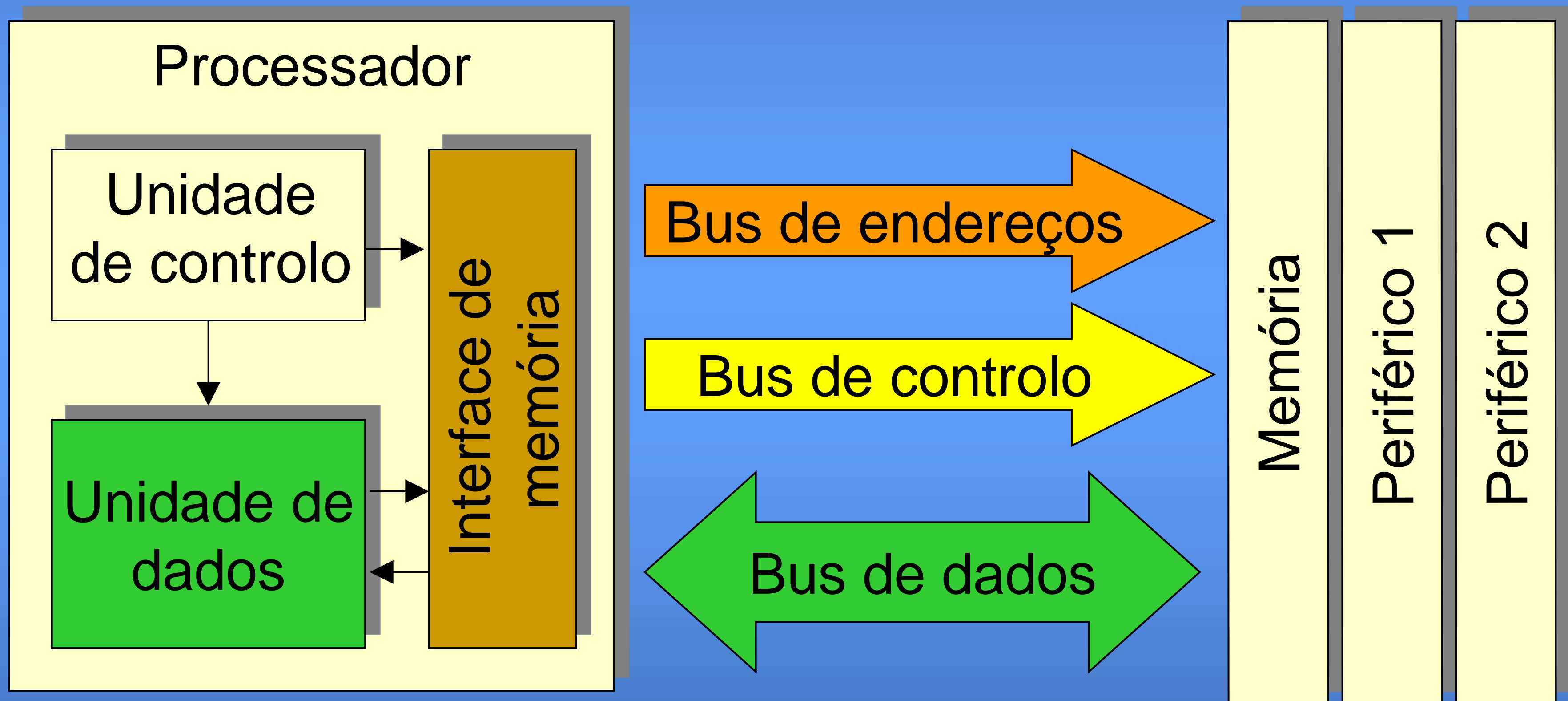
Introdução à programação em linguagem *assembly*

- Espaço de endereçamento
- Instruções de acesso à memória
- Modos de endereçamento
- Diretivas
- Tabelas
- Pilha
- Rotinas

Estrutura de um computador



Espaço de endereçamento

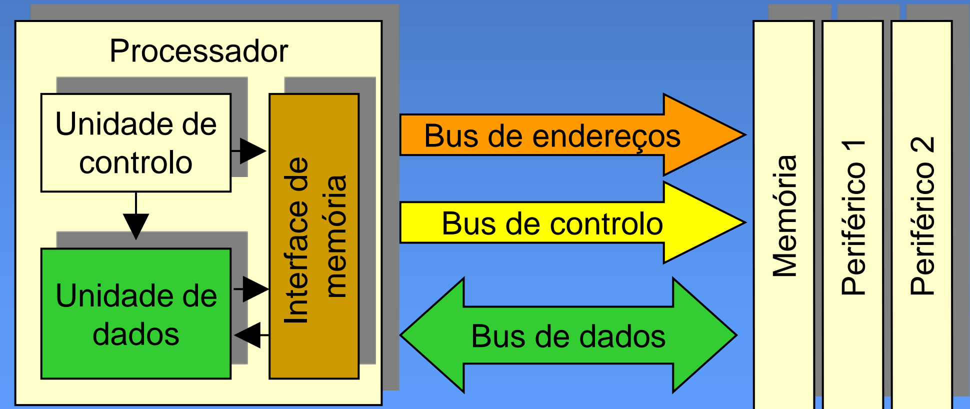
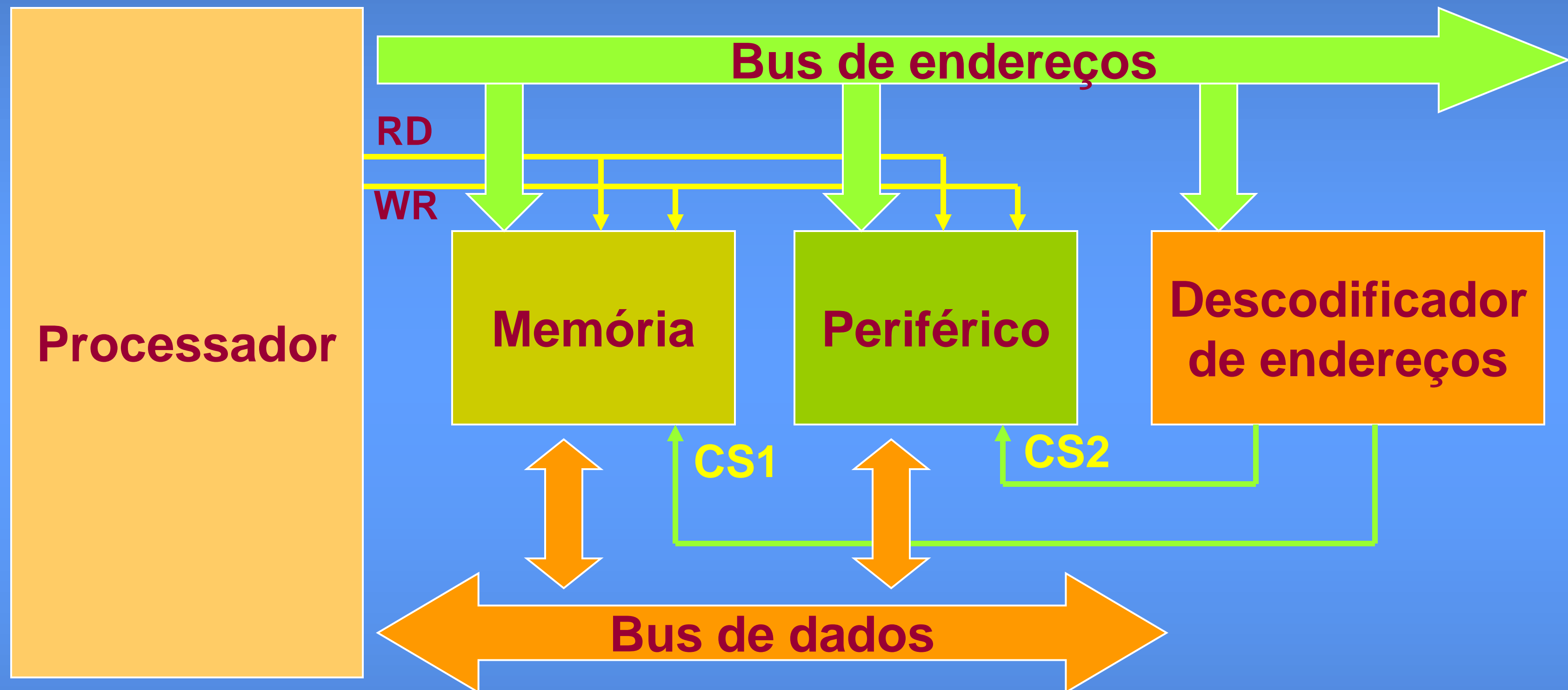
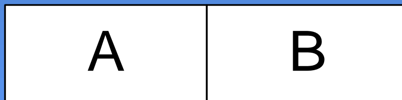
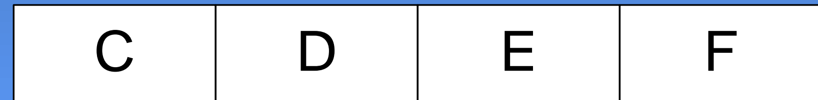


Diagrama de blocos



Endereçamento de byte e de palavra





(a) (b)

Largura do processador	Acessos possíveis à memória, em		
	8 bits	16 bits	32 bits
16 (a)	A B	AB	---
32 (b)	C D E F	CD EF	CDEF

Transferência de dados sem acesso à memória

Instruções	Descrição	Comentários
MOV Rd, Rs	Rd ← Rs	Copia o reg. *Rs* para o reg *Rd*
SWAP Rd, Rs	TEMP ← Rd Rd ← Rs Rs ← TEMP	Troca dois registos TEMP = registo temporário
MOV Rd, k	Rd ← k	Coloca a constante k em Rd k ∈ [-32768 .. 65535]

Acesso à memória (em palavra e em byte)

Instruções	Descrição	Comentários
MOV Rd, [Rs+*off*]	Rd ← M[Rs+*off*]	off ∈ [-16..+14], só pares
MOV Rd, [Rs+Ri]	Rd ← M[Rs+Ri]	Lê 16 bits
MOV [Rd +*off*], Rs	M[Rd +*off*] ← Rs	Escreve 16 bits
MOV [Rd +Ri], Rs	M[Rd +Ri] ← Rs	
MOVB Rd, [Rs]	Rd ← 00H || Mb[Rs]	Só um byte é lido
MOVB [Rd], Rs	Mb[Rd] ← Rs (7..0)	Só um byte na memória é escrito
SWAP Rd, [Rs]	TEMP ← M[Rs] M[Rs] ← Rd Rd ← TEMP	TEMP = registo temporário

Modos de endereçamento

Modo	Exemplo		Comentário
Implícito	PUSH	R1	Manipula SP implicitamente
Imediato	ADD	R1, 3	Só entre -8 e +7
Registo	ADD	R1, R2	
Direto	MOV	R1, [1000H]	**Não suportado pelo PEPE**
Indireto	MOV	R1, [R2]	
Baseado	MOV	R1, [R2 + 6]	
Indexado	MOV	R1, [R2 + R3]	
Relativo	JMP	*etiqueta*	Só dá para aprox. PC $\pm 2^{12}$

- O PEPE não suporta endereçamento direto (porque isso implicava especificar uma constante de 16 bits).

Pseudo-instruções (diretivas)

- São diretivas para o assembler e não instruções para o microprocessador. Logo, não geram código executável.
- Pseudo-instruções típicas:
  - PLACE
  - EQU
  - WORD
  - STRING
  - TABLE

PLACE

- Sintaxe:
  PLACE *endereço*

```
        PLACE 1000H               ; não gera código
1000H   inicio:  MOV    R1, R2    ; "inicio" fica a valer 1000H
1002H            ADD    R1, R3
1004H            CMP    R2, R3
1006H            JZ     inicio    ; salta para "inicio" se R2=R3
1008H            AND    R1, R4
. . .                   . . .
```

EQU

- Sintaxe:
  símbolo EQU *constante-literal*

- Exemplo:

```
DUZIA    EQU   12        ; definição
MOV      R1, DUZIA       ; utilização (R1 ← 12)
```

WORD

- Sintaxe:

  etiqueta: WORD *constante*

- Exemplo:

```
VAR1:    WORD    1    ; variável inicializada a 1.
                      ; Está localizada no endereço
                      ; atribuído pelo assemblador a VAR1
```

WORD é diferente de EQU!

```
        PLACE  1000H       ; início dos endereços gerados
                           ; pelo assemblador
OLA     EQU    4           ; constante definida com o valor 4 (não
                           ; “gasta” endereços do assemblador!)
VAR1:   WORD   1           ; reserva uma palavra de memória, localizada
                           ; no endereço 1000H (valor de VAR1) e
                           ; inicializa-a com 0001H
VAR2:   WORD   OLA         ; Idem, no endereço 1002H (valor de VAR2) e
                           ; inicializa-a com 4 (valor de OLA)
        MOV    R1, OLA     ; R1 ← 4 (isto é uma constante de dados)
        MOV    R2, VAR2    ; R2 ← 1002H (isto é um endereço)
```

Acesso à memória do WORD

```
        PLACE   1000H       ; início dos endereços
OLA     EQU     4           ; constante definida com o valor 4
VAR1:   WORD    1           ; reserva uma palavra no endereço 1000H
VAR2:   WORD    OLA         ; reserva uma palavra no endereço 1002H
        MOV     R1, OLA     ; R1 ← 4 (isto é um dado)
        MOV     R2, VAR2    ; R2 ← 1002H (isto é um endereço)
                            ; isto NÃO acede à memória!

    ; agora sim, vamos aceder à memória
        MOV     R3, [R2]    ; R3 ← M[VAR2], ou
                            ; R3 ← M[1002H]
                            ; R3 fica com 4 (valor do OLA)
        MOV     R4, 0
        MOV     [R2], R4    ; M[VAR2] ← 0, ou
                            ; M[1002H] ← 0
```

Endereço de arranque do PEPE

- Após o *reset*, o PEPE inicializa o PC (endereço de arranque) com o valor 0000H.
- Por isso, tem de haver um PLACE 0000H algures no programa (não tem que ser no início).

```
         PLACE   1000H          ; início dos endereços dos dados

OLA      EQU     4              ; constante definida com o valor 4
VAR1:    WORD    1              ; reserva uma palavra no endereço 1000H
VAR2:    WORD    OLA            ; reserva uma palavra no endereço 1002H

         PLACE   0000H          ; início dos endereços das instruções

Inicio:  MOV     R1, OLA        ; R1 ← 4 (isto é um dado)
         MOV     R2, VAR2       ; R2 ← 1002H (isto é um endereço)
         . . .                  ; resto do programa
```

TABLE

- Sintaxe:

  etiqueta: TABLE *constante*

- Exemplo:

```
T1:    TABLE    10    ; reserva espaço para 10 palavras.
                      ; A primeira fica localizada no
                      ; endereço atribuído a T1, a
                      ; segunda em T1 + 2
```

STRING

- Sintaxe:

  etiqueta: STRING *constante* {, *constante*}

- Exemplo (gasta 5 bytes):

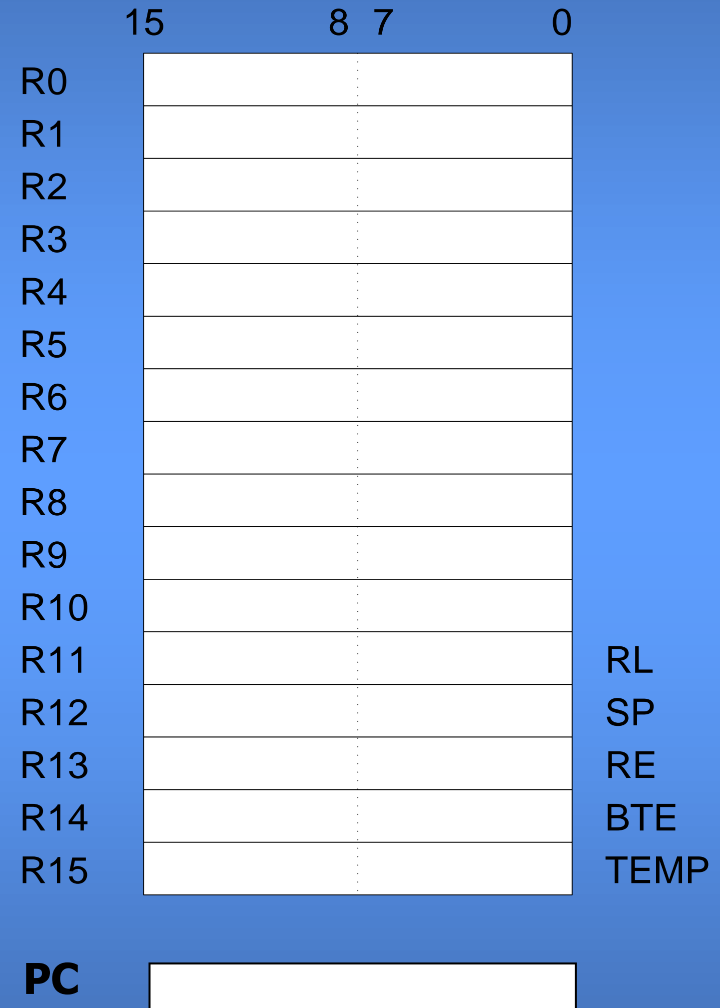
  S1: STRING 'a', “ola”, 12H ; lista de caracteres

Classes de instruções

Classe de instruções	Descrição e exemplos
Instruções aritméticas	Lidam com números em complemento para 2 ADD, SUB, CMP, MUL, DIV
Instruções de bit	Lidam com sequências de bits AND, OR, SET, SHR, ROL
Instruções de transferência de dados	Transferem dados entre dois registos ou entre um registo e a memória MOV, SWAP
Instruções de controlo de fluxo	Controlam a sequência de execução de instruções, podendo tomar decisões JMP, JZ, JNZ, CALL, RET

Registos do processador

- Os recursos mais importantes que as instruções manipulam são os registos.
- O PEPE tem os seguintes registos (todos de 16 bits):
  - PC (Program Counter);
  - 16 registos (R0 a R15), sendo alguns especiais (a ver mais tarde)
- Os registos são uma memória interna, de acesso mais rápido que a externa e com instruções que os manipulam diretamente (mas são muito poucos).



Bits de estado (*flags*)

- Fazem parte do Registo de Estado (RE).
- Fornecem indicações sobre o resultado da operação anterior (nem todas as instruções os alteram).
- Podem influenciar o resultado da operação seguinte.

Bit de estado mais importantes:	Fica a 1 se o resultado de uma operação:
(Z) Zero	for zero
(C) Transporte (*carry*)	tiver transporte
(V) Excesso (*overflow*)	não couber na palavra do processador
(N) Negativo	for negativo

Instruções aritméticas típicas

- Implementam as operações aritméticas das linguagens de alto nível (+, -, *, /).

Instrução	Descrição	Bits de estado afetados
ADD Rd, Rs	Rd ← Rd + Rs	Z, C, V, N
ADDC Rd, Rs	Rd ← Rd + Rs + C	Z, C, V, N
SUB Rd, Rs	Rd ← Rd – Rs	Z, C, V, N
SUBB Rd, Rs	Rd ← Rd – Rs – C	Z, C, V, N
CMP Rd, Rs	Z, C, N, V ← Rd – Rs	Z, C, V, N
MUL Rd, Rs	Rd ← Rd * Rs	Z, C, V, N
DIV Rd, Rs	Rd ← quociente (Rd / Rs)	Z, C, V, N
MOD Rd, Rs	Rd ← resto (Rd / Rs)	Z, C, V, N
NEG Rd	Rd ← – Rd	Z, C, V, N

Instruções de transferência de dados

- Transferem dados entre um registo e:
  - outro registo, ou
  - uma constante (esta variante é só num sentido)
  - uma célula de memória de acesso aleatório, ou
  - uma célula de memória da pilha
- Não afetam nenhum bit de estado (pois não transformam os valores dos dados)

Registos e constantes

Instruções	Descrição	Comentários
MOV Rd, Rs	Rd ← Rs	Copia o reg. *Rs* para o reg *Rd*
SWAP Rd, Rs	TEMP ← Rd Rd ← Rs Rs ← TEMP	Troca dois registos TEMP = registo temporário
MOV Rd, k	Rd ← k	Coloca a constante k em Rd k ∈ [-32768 .. 65535]

- MOV não pode ser só uma instrução (cada instrução tem de ser codificada em apenas 16 bits).
- Solução: construir a constante com duas instruções, MOVL e MOVH, um byte de cada vez (MOV neste caso actua como uma pseudo-instrução).

Construção de constantes

- Consegue apenas especificar-se uma constante de 8 bits com uma instrução.
- Para constantes maiores, é preciso 2 instruções.

Instruções	Descrição	Comentários
MOVL Rd, k	Rd(7..0) ← k Rd(15..8) ← {8}k(7)	k ∈ [0 .. 255] (com extensão de sinal)
MOVH Rd, k	Rd(15..8) ← k	k ∈ [0 .. 255] (só afeta byte de maior peso)
MOV Rd, k	MOVL Rd, k	k ∈ [-128 .. +127] (com extensão de sinal)
MOV Rd, k	MOVL Rd, k(7..0) MOVH Rd, k(15..8)	k ∈ [-32768 .. -129, +128 .. +32767]

Acesso à memória

Instruções	Descrição	Comentários
MOV Rd, [Rs+*off*]	Rd ← M[Rs+*off*]	off ∈ [-16..+14], só pares
MOV Rd, [Rs+Ri]	Rd ← M[Rs+Ri]	Lê 16 bits
MOV [Rd +*off*], Rs	M[Rd +*off]* ← Rs	Escreve 16 bits
MOV [Rd +Ri], Rs	M[Rd +Ri] ← Rs	
MOVB Rd, [Rs]	Rd ← 00H || Mb[Rs]	Só um byte é lido
MOVB [Rd], Rs	Mb[Rd] ← Rs (7..0)	Só um byte na memória é escrito
SWAP Rd, [Rs]	TEMP ← M[Rs] M[Rs] ← Rd Rd ← TEMP	TEMP = registo temporário

Correspondência com as linguagens de alto nível (C)

- Em C:

```
a = 2;  /* variáveis. O compilador escolhe se as ... */
b = a;  /* ... coloca em registos ou na memória */
```

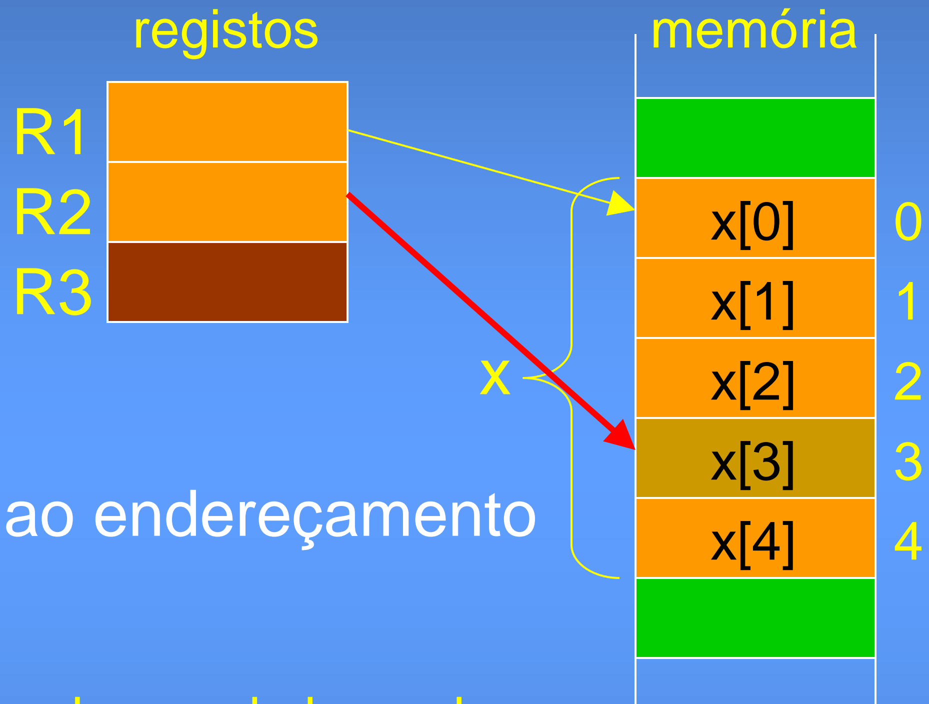
- Em *assembly*, em registos:

```
MOV     R1, 2          ; a = 2 (atribuição)
MOV     R2, R1         ; b = a (atribuição)
```

- Em *assembly*, em memória (mais complexo):

```
MOV     R1, 2
MOV     R2, A          ; A é o endereço da variável a
MOV     [R2], R1       ; a = 2 (escrita na memória)
MOV     R3, B          ; B é o endereço da variável b
MOV     R1, [R2]       ; lê a da memória para um registo
MOV     [R3], R1       ; b = a (escrita na memória)
```

Vetores (*arrays*) em assembly



- Em C:

```
int x[5];
x[3] = x[3] + 2;
```

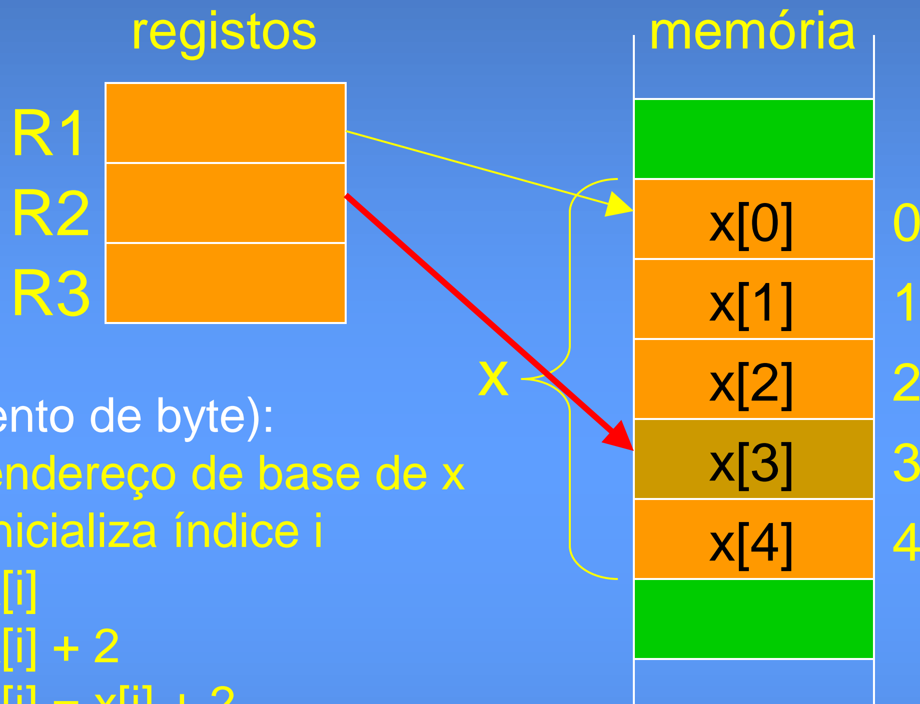
- Em *assembly* (atenção ao endereçamento de byte!):

```
MOV     R1, X          ; endereço de base de x
MOV     R2, [R1+6]     ; x[3]
ADD     R2, 2          ; x[3] + 2
MOV     [R1+6], R2     ; x[3] = x[3] + 2
```

Vetores com índice variável

- Em C:

```
int x[5];     int i;
for (i=0; i!=5 ;i++)
    x[i] = x[i] + 2;
```



- Em *assembly* (com endereçamento de byte):

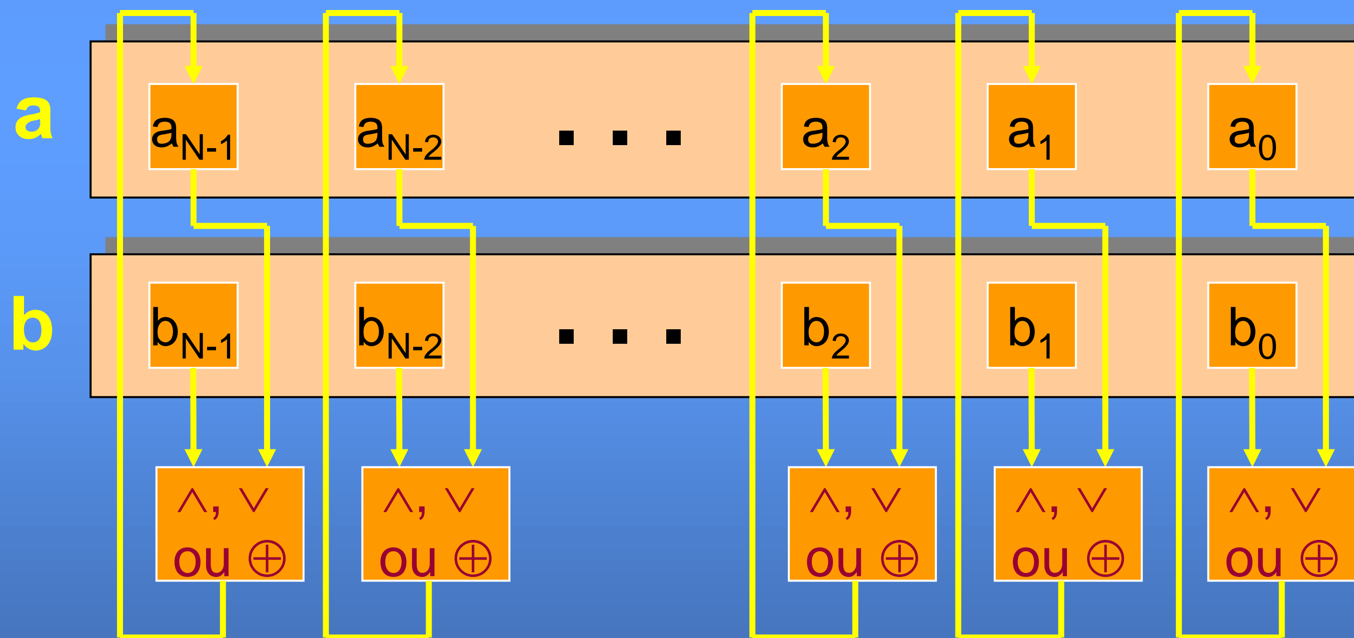
```
    MOV   R1, X          ; endereço de base de x
    MOV   R3, 0          ; inicializa índice i
L1: MOV   R2, [R1+R3]    ; x[i]
    ADD   R2, 2          ; x[i] + 2
    MOV   [R1+R3], R2    ; x[i] = x[i] + 2
    ADD   R3, 2          ; i++
    MOV   R4, 10
    CMP   R3, R4         ; i != 5 (10 em endereço)
    JNZ   L1             ; volta para trás enquanto i!=5
    . . .                ; instruções a seguir ao for
```

Instruções de manipulação de bits

Instrução	Descrição	Bits de estado
TEST Rd, Rs	Z, N $\leftarrow$ Rd $\wedge$ Rs	Z, N
AND Rd, Rs	Rd $\leftarrow$ Rd $\wedge$ Rs	Z, N
OR Rd, Rs	Rd $\leftarrow$ Rd $\vee$ Rs	Z, N
XOR Rd, Rs	Rd $\leftarrow$ Rd $\oplus$ Rs	Z, N
NOT Rd	Rd $\leftarrow$ Rd $\oplus$ FFFFH	Z, N
SHL R, n	n * $[R_{i+1} \leftarrow R_i \ (i \in 0..N\text{-}2); R_0 \leftarrow 0]$	Z, C, N
SHR R, n	n * $[R_i \leftarrow R_{i+1} \ (i \in 0..N\text{-}2); R_{N-1} \leftarrow 0]$	Z, C, N
SHRA R, n	n * $[R_i \leftarrow R_{i+1} \ (i \in 0..N\text{-}2); R_{N-1} \leftarrow R_{N-1}]$	Z, C, N, V
ROL R, n	n * $[R_{i+1} \leftarrow R_i \ (i \in 0..N\text{-}2); R_0 \leftarrow R_{N-1}]$	Z, C, N
ROR R, n	n * $[R_i \leftarrow R_{i+1} \ (i \in 0..N\text{-}2); R_{N-1} \leftarrow R_0]$	Z, C, N
ROLC R, n	n * $[R_{i+1} \leftarrow R_i \ (i \in 0..N\text{-}2); R_0 \leftarrow C; C \leftarrow R_{N-1}]$	Z, C, N
RORC R, n	n * $[R_i \leftarrow R_{i+1} \ (i \in 0..N\text{-}2); R_{N-1} \leftarrow C; C \leftarrow R_0]$	Z, C, N

Instruções lógicas em *assembly*

AND	a, b	$a_i \leftarrow a_i \wedge b_i$ $(i \in 0..N-1)$
OR	a, b	$a_i \leftarrow a_i \vee b_i$ $(i \in 0..N-1)$
XOR	a, b	$a_i \leftarrow a_i \oplus b_i$ $(i \in 0..N-1)$
NOT	a	$a_i \leftarrow \overline{a_i}$ $(i \in 0..N-1)$



Utilização das instruções lógicas

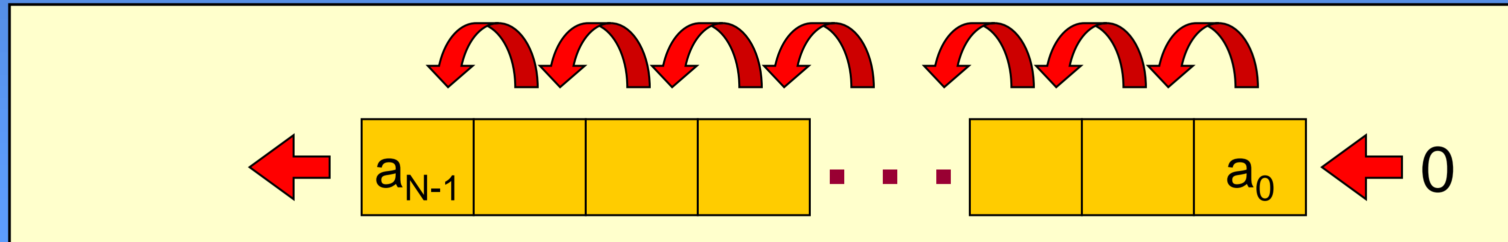
AND	a, b	$a_i \leftarrow a_i \wedge b_i$ $(i \in 0..N-1)$
OR	a, b	$a_i \leftarrow a_i \vee b_i$ $(i \in 0..N-1)$
XOR	a, b	$a_i \leftarrow a_i \oplus b_i$ $(i \in 0..N-1)$
NOT	a	$a_i \leftarrow \overline{a_i}$ $(i \in 0..N-1)$

- Utilizações típicas: forçar bits, teste de bits.

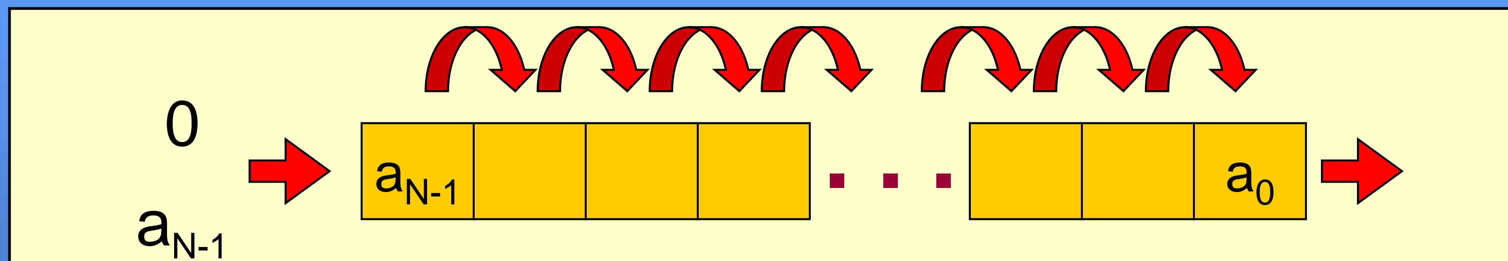
```
; teste de bits no R1 (sem alterar o R1)
        MOV     R2, 09H         ; máscara 0000 1001
        AND     R2, R1          ; isola bits 0 e 3 (sem alterar R1)
        JZ      outroSitio      ; salta se ambos os bits forem 0
; força bits a 1 / troca bits no R1 (alterando o R1)
        MOV     R2, 14H         ; máscara 0001 0100
        OR      R1, R2          ; força bits 2 e 4 a 1 (troca se XOR)
```

Instruções de deslocamento

SHL	a, n	n * [$a_{i+1} \leftarrow a_i$ ($i \in 0..N-2$); $a_0 \leftarrow 0$]



SHR	a, n	n * [$a_i \leftarrow a_{i+1}$ ($i \in 0..N-2$); $a_{N-1} \leftarrow 0$]
SHRA	a, n	n * [$a_i \leftarrow a_{i+1}$ ($i \in 0..N-2$); $a_{N-1} \leftarrow a_{N-1}$]



- Correspondem a multiplicar e dividir por 2^n.

Utilização das instruções de deslocamento

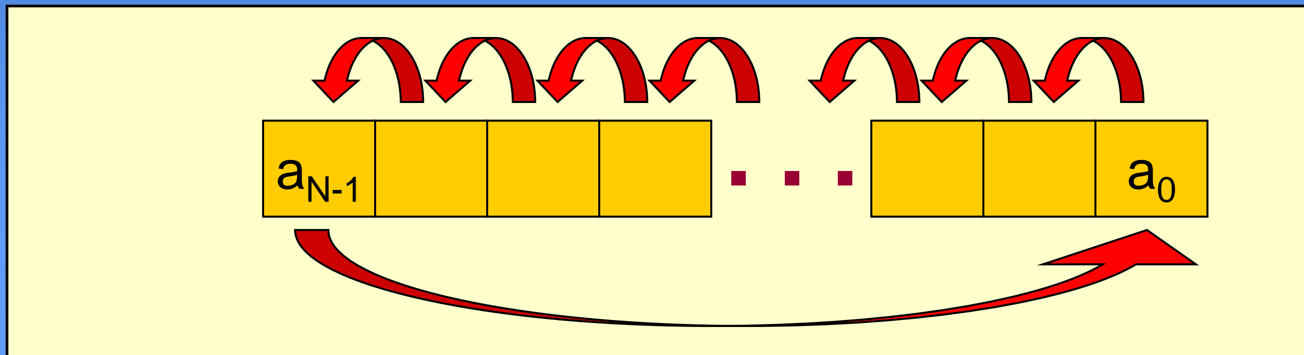
SHL	a, n	n * [a_{i+1} ← a_i (i ∈ 0..N-2); a_0 ← 0]
SHR	a, n	n * [a_i ← a_{i+1} (i ∈ 0..N-2); a_{N-1} ← 0]
SHRA	a, n	n * [a_i ← a_{i+1} (i ∈ 0..N-2); a_{N-1} ← a_{N-1}]

- Utilizações típicas: mudar bits de sítio

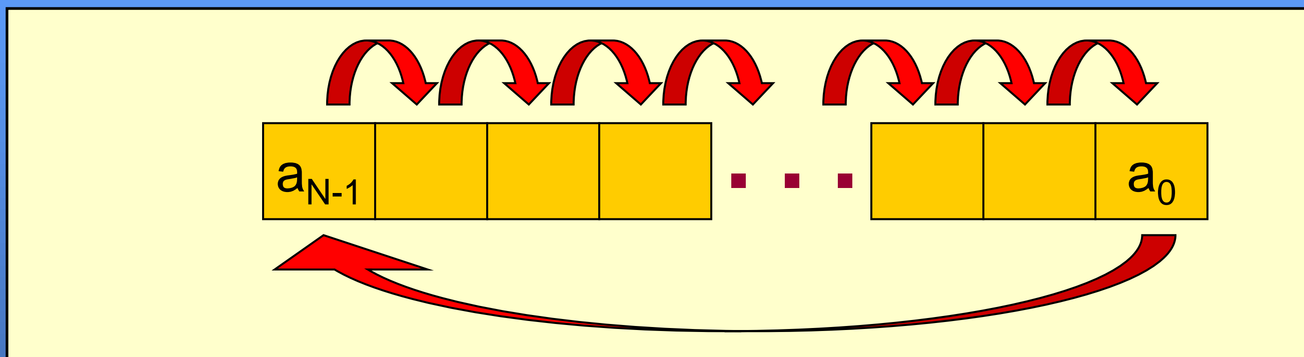
```
; isolar e juntar os bits 7 e 1 do R1, por esta ordem, nos bits 5 e 4 de R2
; com os outros bits a zero (7 → 5 e 1 → 4)
        MOV     R2, 80H     ; máscara 1000 0000
        AND     R2, R1      ; isola bit 7 de R1 em R2
        SHR     R2, 2       ; 2 casas para a direita (7 → 5)
        MOV     R3, 02H     ; máscara 0000 0010
        AND     R3, R1      ; isola bit 1 de R1 em R3
        SHL     R3, 3       ; 3 casas para a esquerda (1 → 4)
        OR      R2, R3      ; junta as duas partes
```

Instruções de rotação

ROL a, n	$n * [a_{i+1} \leftarrow a_i \ (i \in 0..N\text{-}2); \ a_0 \leftarrow a_{N-1}]$



ROR a, n	$n * [a_i \leftarrow a_{i+1} \ (i \in 0..N\text{-}2); \ a_{N-1} \leftarrow a_0]$



Utilização das instruções de rotação

ROL	a, n	n * [$a_{i+1} \leftarrow a_i$ ($i \in 0..N-2$); $a_0 \leftarrow a_{N-1}$]
ROR	a, n	n * [$a_i \leftarrow a_{i+1}$ ($i \in 0..N-2$); $a_{N-1} \leftarrow a_0$]

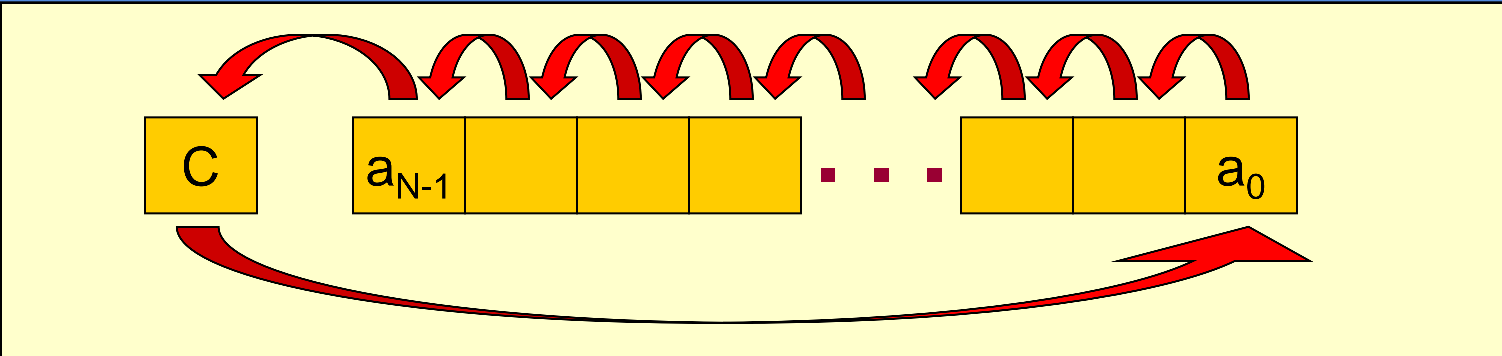
- Utilizações típicas: trocar bits dentro de uma palavra

```
; trocar o byte alto e baixo de registo
        ROL    R1, 8      ; rotação de 8 bits para a esquerda
        ROR    R1, 8      ; rotação de 8 bits para a direita
```

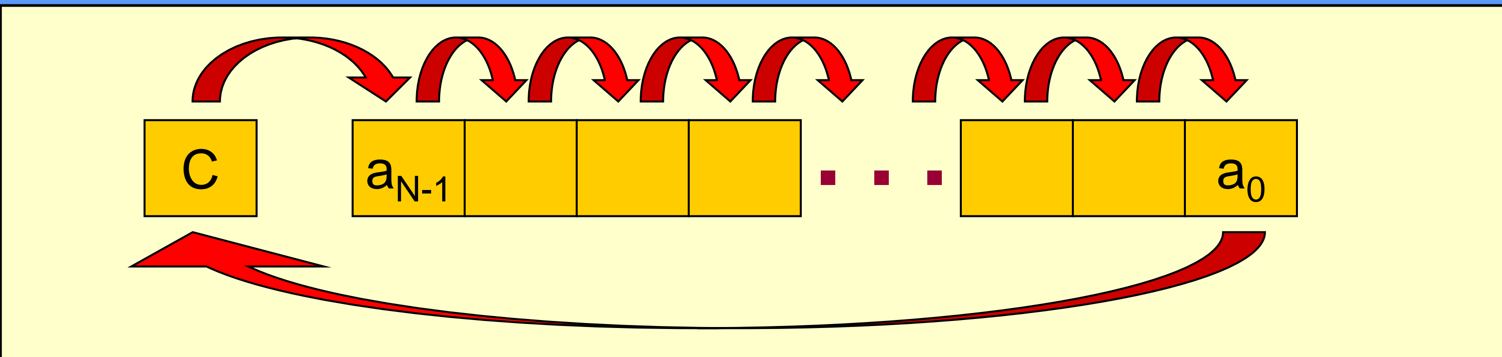
- Neste caso tanto faz rodar para um lado ou para o outro
- Nenhum bit se perde (ao contrário dos deslocamentos)

Instruções de rotação com *carry*

ROLC a, n	n * [$a_{i+1} \leftarrow a_i$ ($i \in 0..N-2$); $a_0 \leftarrow C$; $C \leftarrow a_{N-1}$]



RORC a, n	n * [$a_i \leftarrow a_{i+1}$ ($i \in 0..N-2$); $a_{N-1} \leftarrow C$; $C \leftarrow a_0$]



Exemplo: contagem de bits a 1

```
valor     EQU   6AC5H      ; valor cujos bits a 1 vão ser contados
início:   MOV   R1, valor  ; inicializa registo com o valor a analisar
          MOV   R2, 0      ; inicializa contador de número de bits=1
maisUm:   ADD   R1, 0      ; isto é só para atualizar os bits de estado
          JZ    fim        ; se o valor já é zero, não há mais bits
                           ; a 1 para contar
          SHR   R1, 1      ; retira o bit de menor peso do valor e
                           ; coloca-o no bit C
          MOV   R3, 0      ; ADDC não suporta constantes
          ADDC  R2, R3     ; soma mais 1 ao contador, se esse bit=1
          JMP   maisUm     ; vai analisar o próximo bit
fim:      JMP   fim        ; acabou. Em R2 está o número de bits=1
```

Controlo de fluxo

- A execução das instruções numa linguagem de alto nível é sequencial, exceto quando temos uma:
  - decisão (*if*, *switch*)
  - iteração
    - incondicional – *for*
    - condicional - *while*
  - chamada ou retorno de uma função ou procedimento
- Em *assembly*, o controlo de fluxo é feito com:
  - bits de estado (indicam resultado da instrução anterior)
  - instruções específicas de:
    - salto (condicionais ou incondicionais)
    - chamada de rotina
    - retorno de rotina

Instruções de salto

- São instruções cujo objetivo é alterar o PC (em vez de o deixarem incrementar normalmente).
- Saltos:
  - Incondicionais (ex: JMP *etiqueta*)
  - Condicionais (ex: JZ *etiqueta*)
- Saltos:
  - Absolutos (ex: JMP R1 ---> PC ← R1)
  - Relativos (ex: JMP *etiqueta* ---> PC ← PC + dif)
  - dif = etiqueta – PC (é o que assembler põe na instrução)
  - dif tem apenas 12 bits no JMP e 8 bits no JZ

Saltos relativos

- Programas relocáveis: podem ser localizados em qualquer ponto da memória (só com saltos relativos ao PC atual).
- Os saltos têm a forma "JMP *etiqueta*" apenas para serem mais claros para o programador (vê-se logo para onde vão), mas

  JMP *etiqueta* ---> PC ← PC + dif

  dif = etiqueta – PC (é o que o assembler põe na instrução)
- dif tem apenas 12 bits no JMP e 8 bits nos saltos condicionais (localidade limitada).
- Como as instruções têm de estar em endereços pares, dif tem de ser par, o que o permite estender a 13 e 9 bits (na instrução omite-se o bit de menor peso, que é sempre 0).
- Nota importante: quando uma instrução é executada, o PC já tem o endereço da próxima instrução.
  - dif = 0 é um NOP
  - dif = -2 é um ciclo infinito

Instrução *if*

- Em C:

  if (*expressão-booleana*) /* 0 ≡ false, != 0 ≡ true */
  { *instruções* }

- Em *assembly*:

```
        expressão           ; calcula expressão (afeta bit de estado Z)
        JZ        OUT       ; se expressão booleana for falsa,
                            ; não executa instruções
        instruções          ; código das instruções dentro do if
OUT:    ...                 ; instrução a seguir ao if
```

Instrução *if-else*

- Em C:

  if (*expressão-booleana*)

  { *instruções 1* } else { *instruções 2*}

- Em *assembly*:

```
        expressão       ; calcula expressão (afeta bit de estado Z)
        JZ    ALT       ; se expressão booleana for falsa, salta
        instruções 1    ; código das instruções dentro do if
        JMP   OUT       ; não pode executar instruções 2
ALT:    instruções 2    ; código das instruções da cláusula else
OUT:    ...             ; instrução a seguir ao if
```

Expressões booleanas no *if*

- Para fazer:

  if (*a* < *b*)

  { *instruções* }

- O compilador pode fazer:

```
        CMP     Ra,Rb       ; afeta bit de estado N
        JGE     OUT         ; se a >= b, bit de estado N estará a 0
                            ; ou então: JNN    OUT
        instruções          ; código das instruções dentro do if
OUT:    ...                 ; instrução a seguir ao if
```

- O PEPE tem instruções para suportar os vários casos relacionais possíveis (<, <=, >, >=, =, <>)

Ciclos (iteração)

- Destinam-se a executar um bloco de código um certo número de vezes.
- Fixo, ou incondicional (*for*)

```
for (i=0; i < N; i++;)
    { instruções }
```

- Condicional (*while* e *do-while*)

```
while (expressão)
    { instruções }
```

```
do
    { instruções }
while (expressão);
```

Ciclos incondicionais (*for*)

- Em C:

```
for (i=0; i < N; i++)
    { instruções }
```

- Em *assembly* (assumindo que *i* está no registo R1):

```
        MOV   R1, 0   ; inicializa variável de índice (i = 0;)
LOOP:   MOV   R2, N
        CMP   R1, R2  ; i < N?
        JGE   OUT     ; se i>=N, já terminou e vai embora
        instruções    ; código das instruções dentro do for
        ADD   R1, 1   ; i++
        JMP   LOOP    ; próxima iteração
OUT:    ...           ; instrução a seguir ao for
```

Ciclos condicionais (*while*)

- Em C:

```
while (expressão)
    { instruções }
```

- Em *assembly*:

```
LOOP:   expressão           ; código para calcular a expressão
        JZ      OUT         ; se expressão for falsa, sai do ciclo
        instruções          ; código das instruções dentro do while
        JMP     LOOP        ; próxima iteração (avalia expressão de novo)
OUT:    ...                 ; instrução a seguir ao while
```

Ciclos condicionais (*do-while*)

- Em C:

  do
  { *instruções* }
  while (*expressão*);

- Em *assembly*:

LOOP:	*instruções*		; instruções dentro do *do-while*
	expressão		; instruções para calcular *expressão*
	JNZ	LOOP	; se *expressão* for verdadeira, continua no ciclo
OUT:	...		; instrução a seguir ao *do-while*

Exercícios

1. Considere os dois casos seguintes de linhas de *assembly*:

```
PLACE   1000H
WORD    1234H
```

e

```
PLACE   0000H
MOV     R1, 1234H
MOV     R2, 1000H
MOV     [R2], R1
```

a) Indique para cada caso o valor com que fica a posição de memória com endereço 1000H;
b) Explique as diferenças entre os dois casos.

Exercícios

2. O PEPE não tem uma instrução para fazer deslocamento de um número variável de bits (ex: SHR R1, R2). Implemente um programa que desloque o R1 à direita de um número de bits indicado por R2.
3. Faça um programa que conte o número de caracteres de uma string em C (1 byte/carácter, terminada por 0).
4. Usando as instruções DIV (divisão inteira) e MOD (resto da divisão inteira), faça um programa que converte o valor de um registo (em binário) numa string com dígitos em decimal, correspondente ao mesmo valor (ex: 1000H → “4096”). Nota: “0” em ASCII é 30H.
5. Construa um programa para executar a operação inversa, passar de uma string ASCII para um valor inteiro. Assume-se a string terminada por 0 e bem formada (só dígitos, valor dentro dos limites).

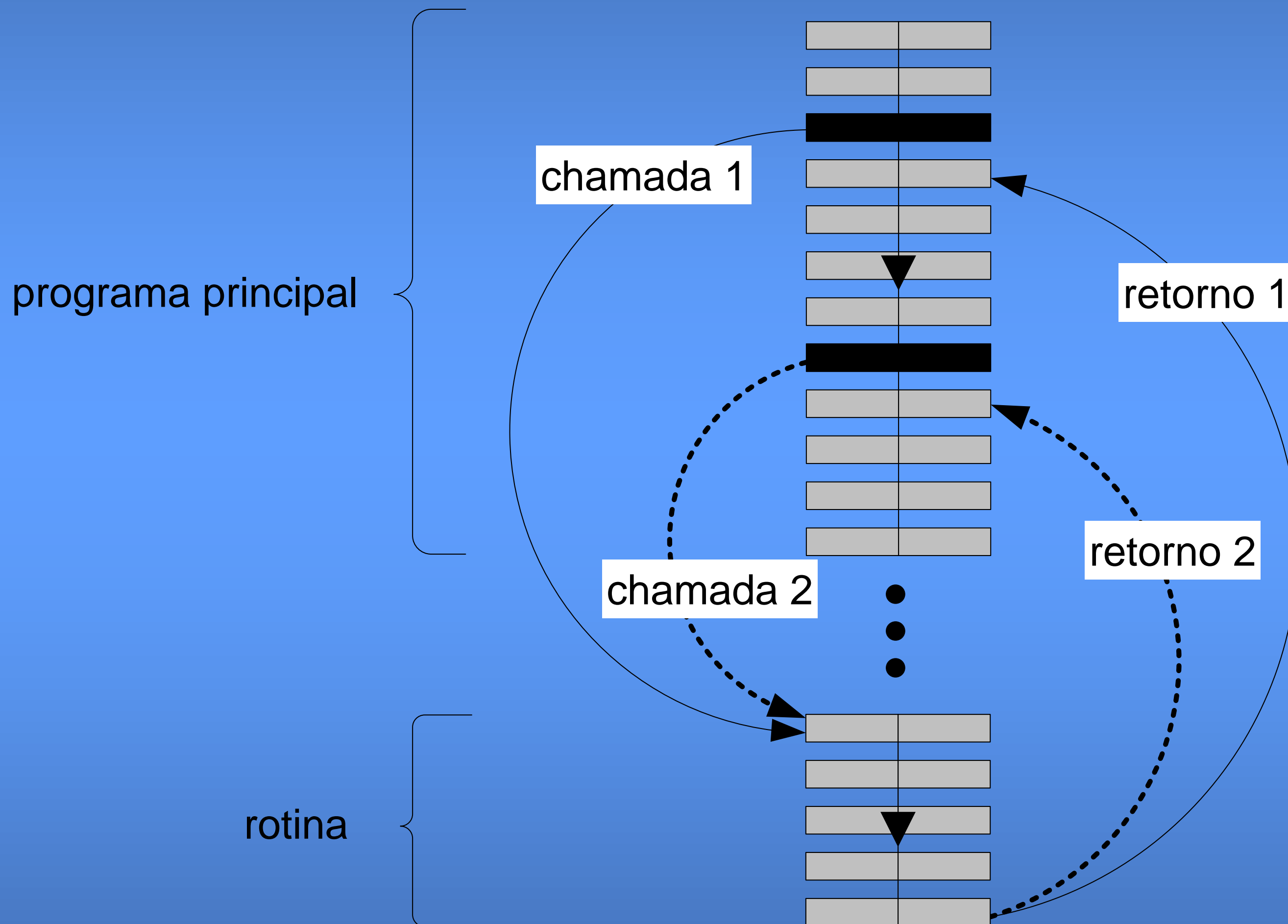
Exercícios

6. Imagine o seguinte código em C, em que se assume que *i* e *total* são variáveis inteiras de 16 bits e A é um array de 10 inteiros de 16 bits cada:

```
total = 0;
for (i = 0; i < 10; i++;)
        total = total + A[i];
```

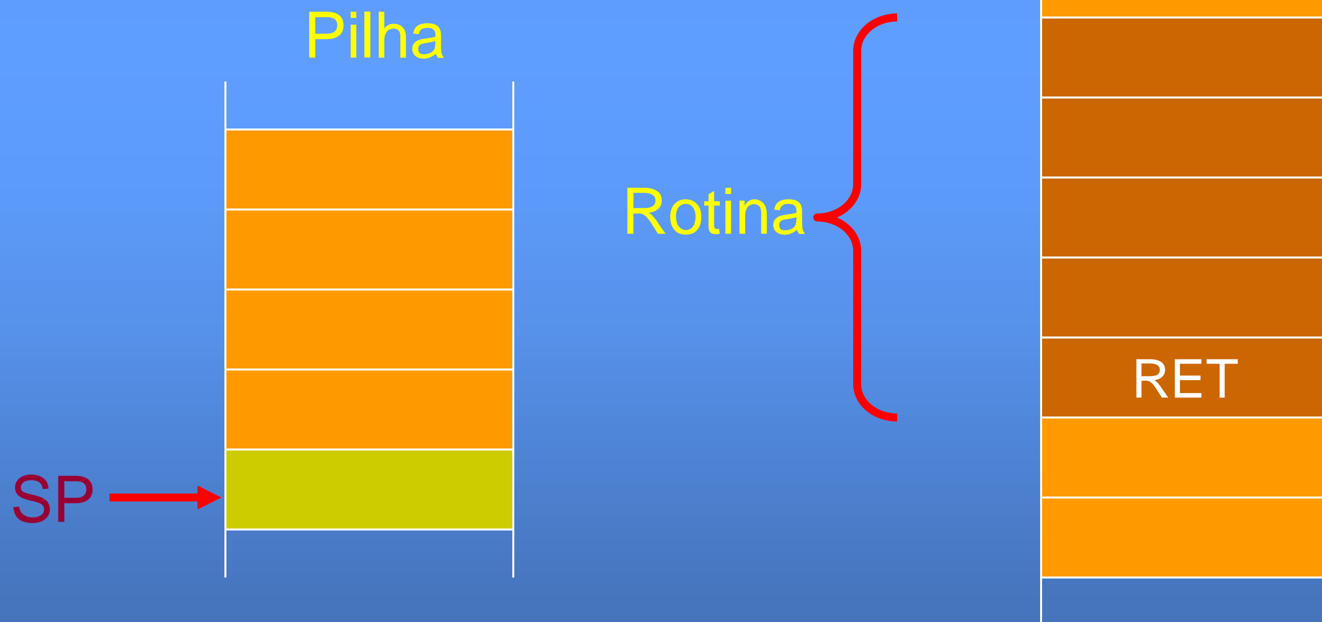
Faça um programa em assembly do PEPE que seja equivalente. Assuma que a variáveis *i* e *total* estão nos registos R0 e R1, respetivamente, e que o array *A* é uma estrutura de dados em memória que começa no endereço A.

Uso de rotinas



Chamada/retorno

- As rotinas não sabem de onde são chamadas.
- O par CALL-RET resolve esta questão automaticamente.



Chamada/retorno

- A pilha memoriza o endereço seguinte ao CALL (valor do PC)
- SP ← SP - 2
- M[SP] ← PC
- PC ← *endereço da rotina*

Pilha

SP

Retorno

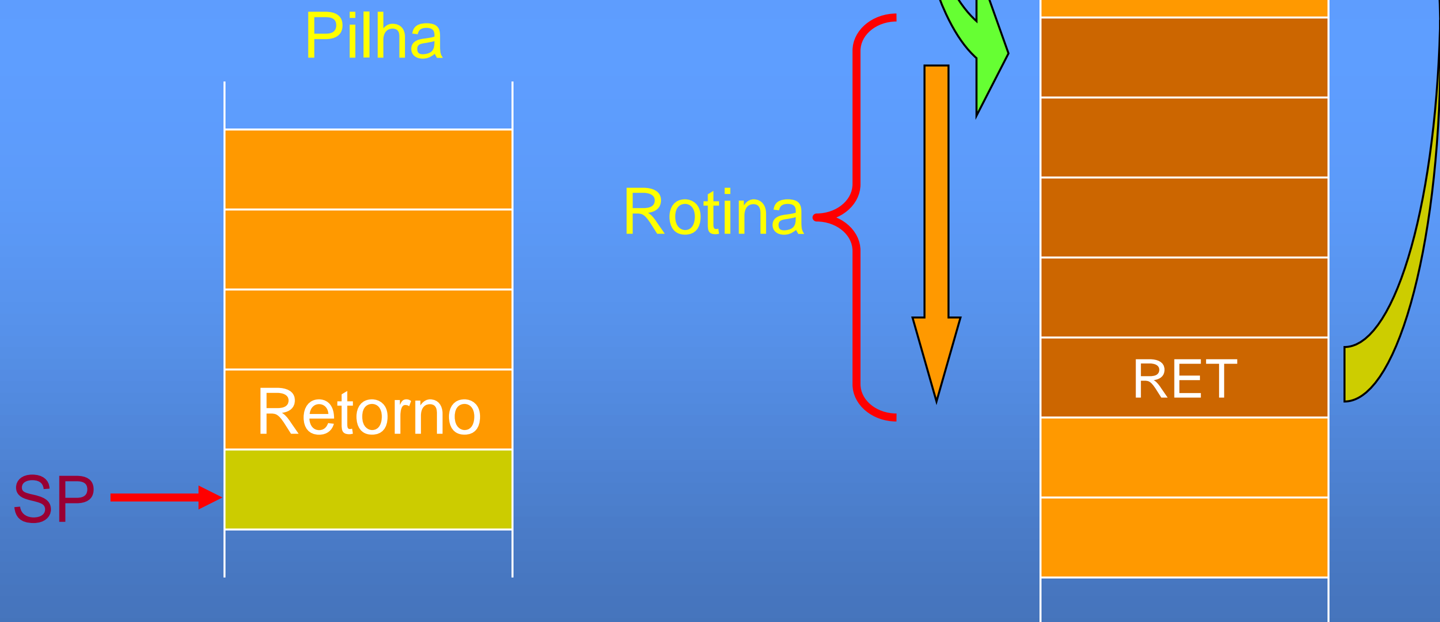
Rotina

CALL

RET

Chamada/retorno

- A pilha memoriza o endereço seguinte ao CALL (valor do PC)
- PC ← M[SP]
- SP ← SP + 2
- RET usa esse endereço para retornar



Rotinas e a pilha

Programa principal	Rotina A	Rotina B	Rotina C	Rotina D	Estado da pilha
					Pilha vazia
CALL → A					end. ret. A
	CALL → B				end. ret. B end. ret. A
		CALL → C			end. ret. C end. ret. B end. ret. A
		← RET			end. ret. B end. ret. A
	← RET				end. ret. A
	CALL → D				end. ret. D end. ret. A
	← RET				end. ret. A
← RET					Pilha vazia

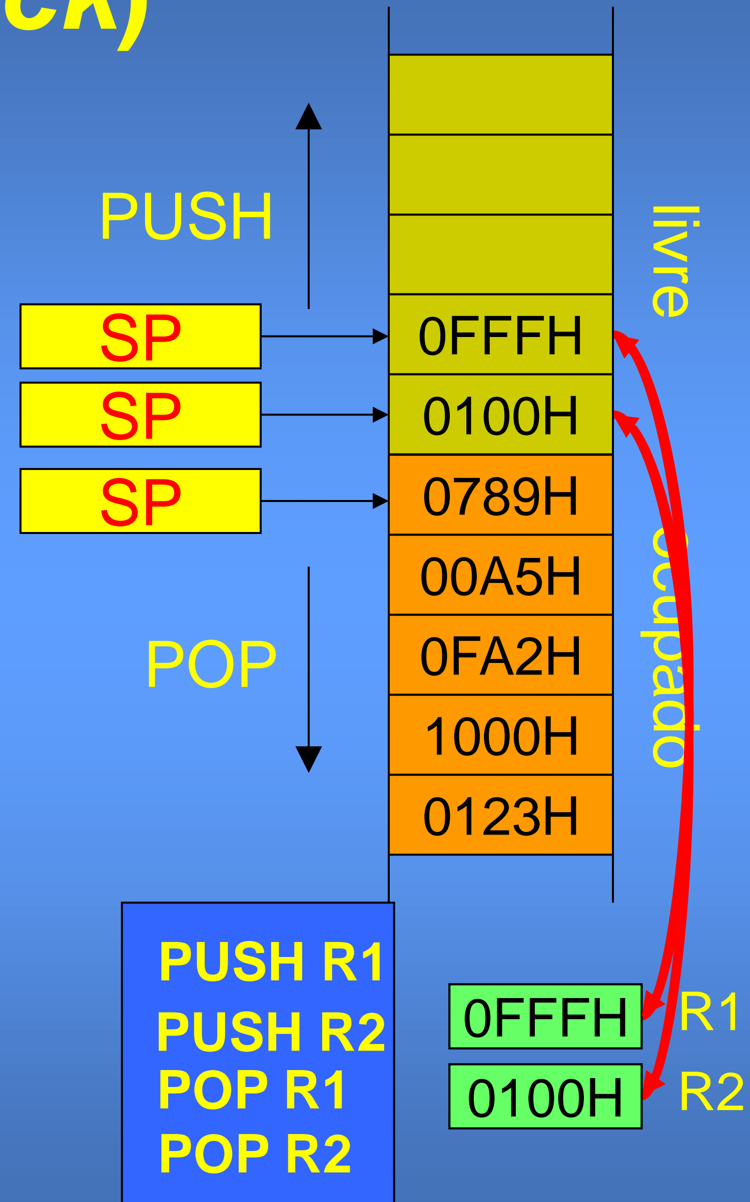
TEMPO

FIM

Pilha (*stack*)

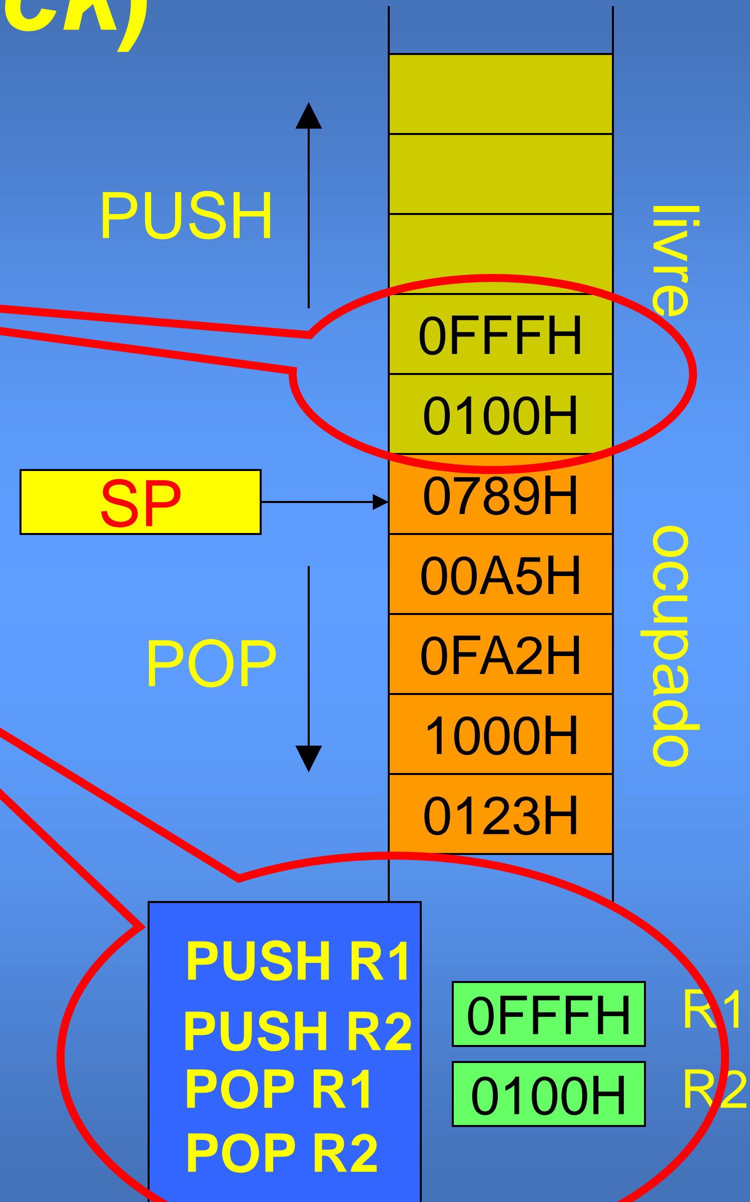
- SP aponta para a última posição ocupada da pilha (topo da pilha)

- **PUSH R_i**: SP←SP-2; M[SP]←R_i
- **POP R_i**: R_i←M[SP]; SP←SP+2

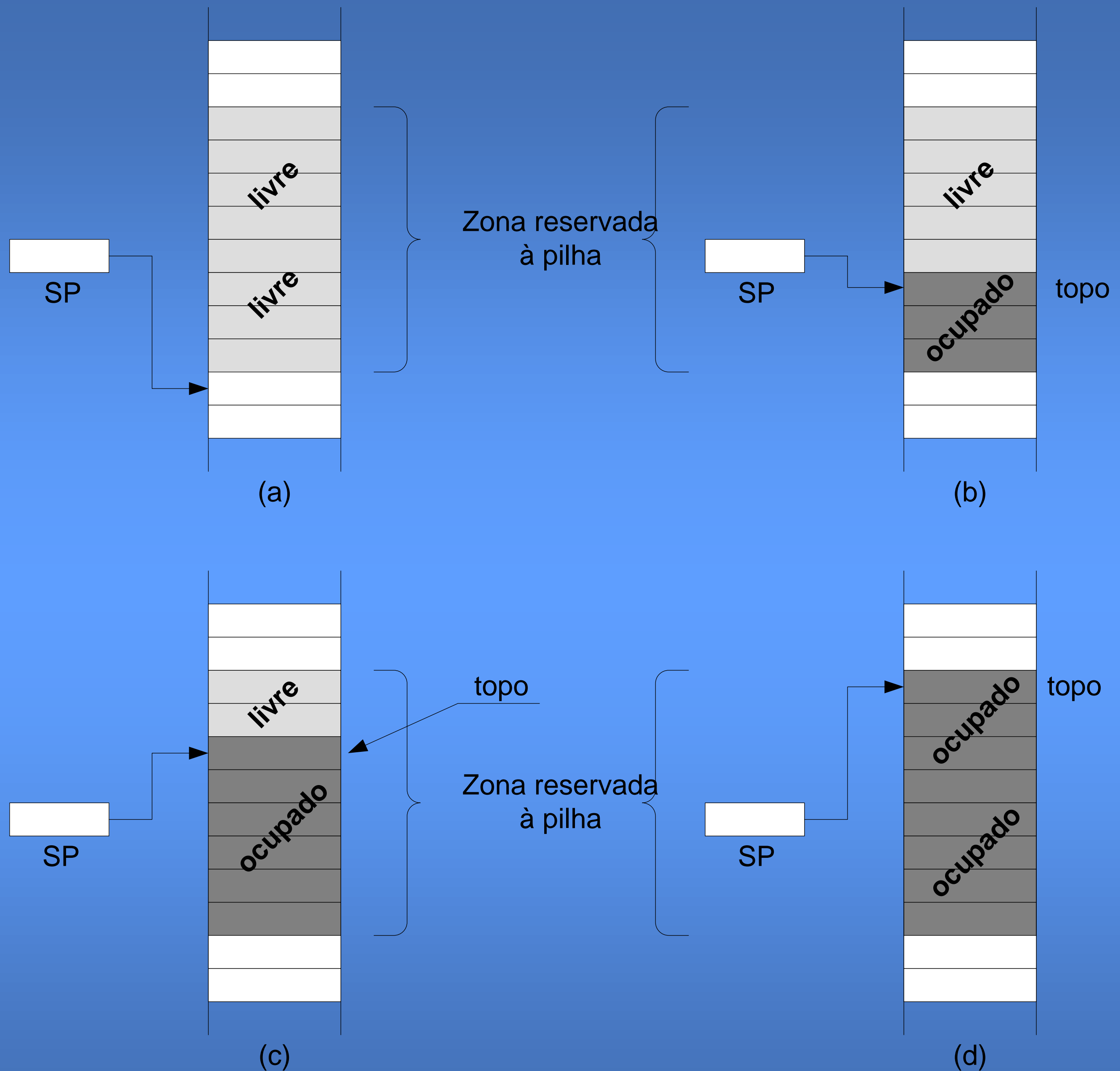


Pilha (*stack*)

- O POP não apaga os valores, apenas os deixa na zona livre.
- Os POPs têm de ser feitos pela ordem inversa dos PUSHes, senão os valores vêm trocados!



Pilha e SP (*Stack Pointer*)



Instruções de chamada/retorno

Instruções	Descrição	Comentários
JMP K	PC ← PC + K	Salto sem retorno
CALL K	SP ← SP - 2 M[SP] ← PC PC ← PC + K	Ajusta SP Guarda endereço de retorno na pilha Salta para a rotina
RET	PC ← M[SP] SP ← SP + 2	Recupera endereço de retorno Ajusta SP

Guardar registos na pilha

Instruções	Descrição	Comentários
PUSH Rs	SP ← SP - 2 M[SP] ← Rs	SP aponta sempre para a última posição ocupada (topo)
POP Rd	Rd ← M[SP] SP ← SP + 2	POP não destrói os valores lidos da pilha

- Antes de se utilizar a pilha tem de se:
  - Inicializar o SP
  - Verificar o tamanho máximo previsível para a pilha e reservar espaço suficiente

Instruções CALL e RET

- A instrução CALL *rotina* equivale conceptualmente a:

```
PUSH    PC        ; guarda o endereço da instrução
                  ; que vem a seguir ao CALL
JMP     rotina    ; transfere controlo para a rotina
```

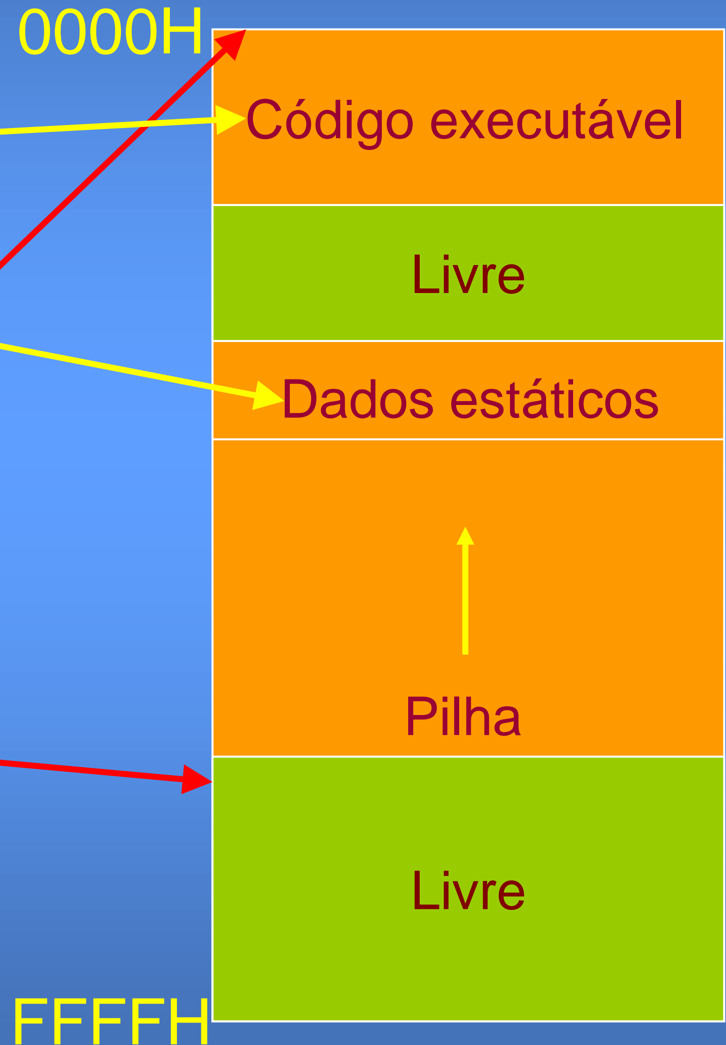
- A instrução RET equivale conceptualmente a :

```
POP     PC        ; retira da pilha o endereço da instrução
                  ; para onde deve retornar e salta para lá
```

- O mecanismo LIFO da pilha garante a imbricação de rotinas (ordem de retorno é inversa da ordem de chamada).

Mapa de endereçamento

- PLACE permite localizar:
  - Blocos de código
  - Dados estáticos (variáveis criadas com WORD)
- No PEPE, o PC é inicializado com 0000H
- A pilha é localizada através da inicialização do SP



Passagem de valores

- Invocação: z = soma (x, y);
- Quem chamar a função tem de colocar os parâmetros num local combinado com a função.
- Idem para o valor de retorno

- Registos

```
MOV     R1, x
MOV     R2, y
CALL    soma
; resultado em R3
```

- Memória (pilha)

```
MOV     R1, x       ; 1º operando
PUSH    R1
MOV     R2, y       ; 2º operando
PUSH    R2
CALL    soma
POP     R3
MOV     R4, z       ; endereço resultado
MOV     [R4], R3
```

Passagem de valores por registos

```
MOV     R1, x
MOV     R2, y
CALL    soma
; resultado em R3
```

- Vantagens
  - Eficiente (registos)
  - Registo de saída de uma função pode ser logo o registo de entrada noutra (não é preciso copiar o valor)
- Desvantagens
  - Menos geral (número de registos limitado, não suporta recursividade)
  - Estraga registos (pode ser preciso guardá-los na pilha)

Passagem de valores pela pilha

```
MOV     R1, x    ; 1º operando
PUSH    R1
MOV     R2, y    ; 2º operando
PUSH    R2
CALL    soma
POP     R3
MOV     R4, z    ; endereço resultado
MOV     [R4], R3
```

- Vantagens
- Genérico (dá para qualquer número de parâmetros)
- Recursividade fácil (já se usa a pilha)
- Desvantagens
  - Pouco eficiente (muitos acessos à memória)
  - É preciso cuidado com os PUSHes e POPs (tem de se "consumir" os parâmetros e os valores de retorno)

Salvaguarda de registos

- Uma rotina nunca sabe de onde é chamada
- Se usar registos, tem de:
  - salvá-los (na pilha) antes de os usar
  - restaurá-los pela ordem inversa antes de retornar

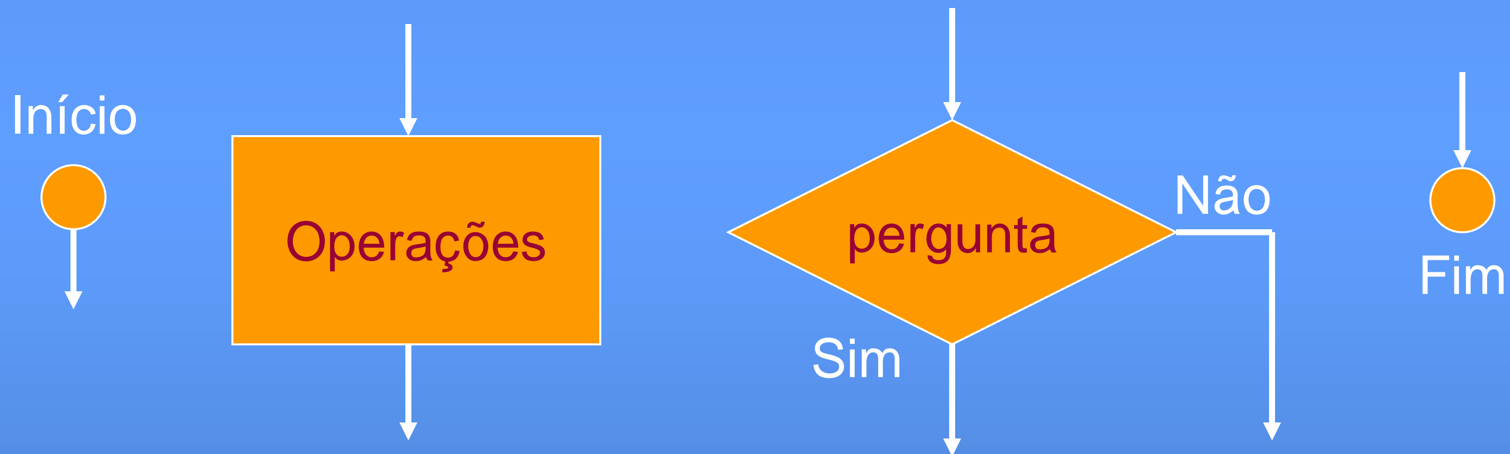
```
. . .
PUSH  R1  ; salva R1
PUSH  R2  ; salva R2
. . .     ; código da rotina que altera R1 e R2
POP   R2  ; restaura R2
POP   R1  ; restaura R1
RET       ; já pode retornar
```

Exemplo de comentários em rotinas

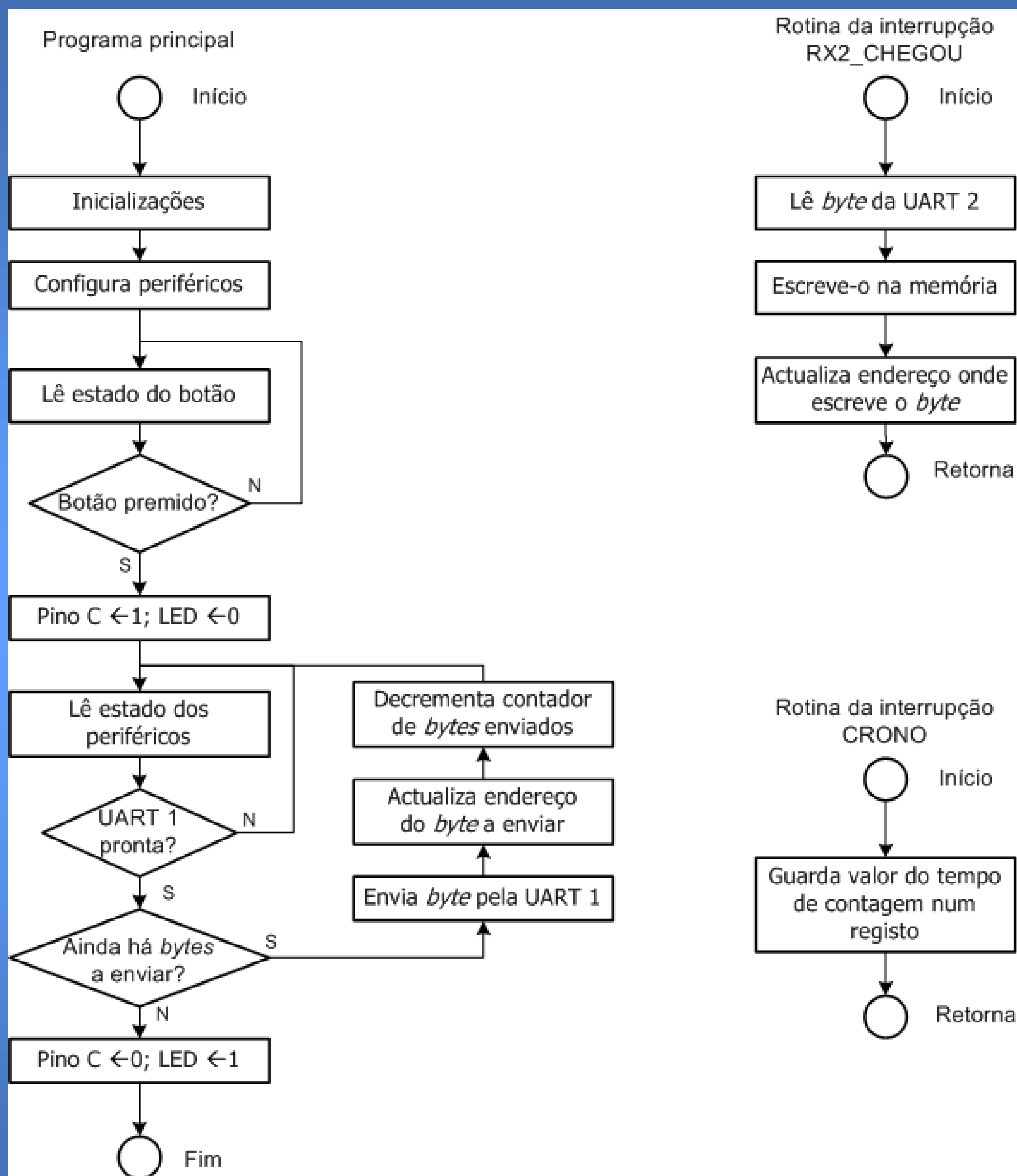
```
;*****************************************************
;    Nome:            FACT
;    Autor:           Eu
;    Alterada em:     Ontem
;    Descrição:       Calcula o factorial de um número (n!)
;    Entradas:        R1 - Parâmetro (valor n)
;    Saídas:          R2 - Factorial de n (n!)
;    Altera:          Bits de estado
;*****************************************************
FACT:   PUSH    R1
        CMP     R1, 1      ; n válido?
        JGE     Ok         ; vai tratar casos n >= 1
        MOV     R2, 1      ; n! = 1 ( se n<1)
Sai:    POP     R1
        RET
Ok:     MOV     R2, R1     ; inicializa resultado com n
Ciclo:  SUB     R1, 1      ; n - 1
        JZ      Sai        ; se R1, já era 1, acabou
        MUL     R2, R1     ; resultado = resultado * n-1
        JMP     Ciclo      ; (vai acumulando)
```

Fluxograma

- Notação gráfica para especificar o comportamento de uma rotina
- Construções fundamentais:



Fluxograma (exemplo: Fig. B.2 do livro)



Exercícios

1. Suponha que nos endereços de memória indicados na tabela seguinte estão as instruções referidas.

1000H	X:	MOV	R1, 0040H
1002H		PUSH	R1
1004H		RET	
2000H		CALL	X

a) Diga qual o significado de X e qual o seu valor;

b) Suponha que o valor inicial do PC é 2000H. Simule a execução do processador, relatando o que se passa em seguida (sugestão: faça uma tabela em que para cada instrução executada indique que registos são alterados, incluindo PC e SP, e quais os novos valores);

c) Quando é que este programa pára (isto é, quando é que o processador executará uma instrução fora da tabela)?

Exercícios

2. Suponha a seguinte sequência de instruções e que o valor inicial dos registos é o indicado:

PUSH	R1	
PUSH	R3	R1 = 1000H
PUSH	R2	R2 = 2000H
POP	R1	R3 = 3000H
POP	R2	SP = 4000H
POP	R3	PC = 5000H

a) Qual o valor final de R1, R2, R3, SP e PC?

b) Qual os valores máximo e mínimo do SP ao longo da sequência?

c) Faça uma pequena tabela com os endereços das células de memória alteradas e os respetivos valores finais.

d) Mude a ordem da sequência de modo que os registos R1, R2 e R3 não tenham os seus valores alterados.

Exercícios

3. Imagine que tem um vetor de números inteiros (16 bits) em memória, em posições consecutivas. A dimensão do vetor não é fixa. O seu fim é indicado pelo primeiro número negativo (que é apenas terminador e já não é elemento do vetor). Pretende-se desenvolver uma rotina em *assembly* que determine qual o maior dos inteiros do vetor. A rotina recebe como parâmetro, no registo R1, o endereço do primeiro inteiro do vetor. O resultado deverá ser retornado no próprio registo R1.

   a) Desenhe um fluxograma que corresponda à função pretendida;
   b) Escreva a rotina em linguagem *assembly* do processador PEPE.