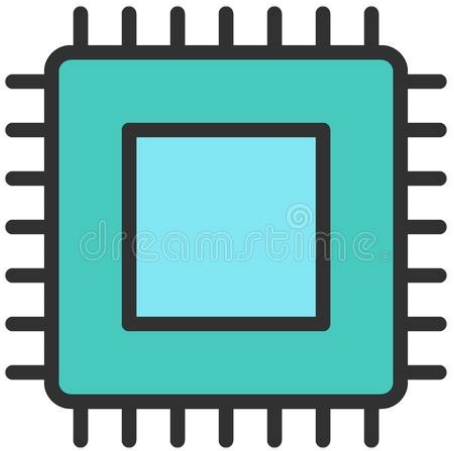
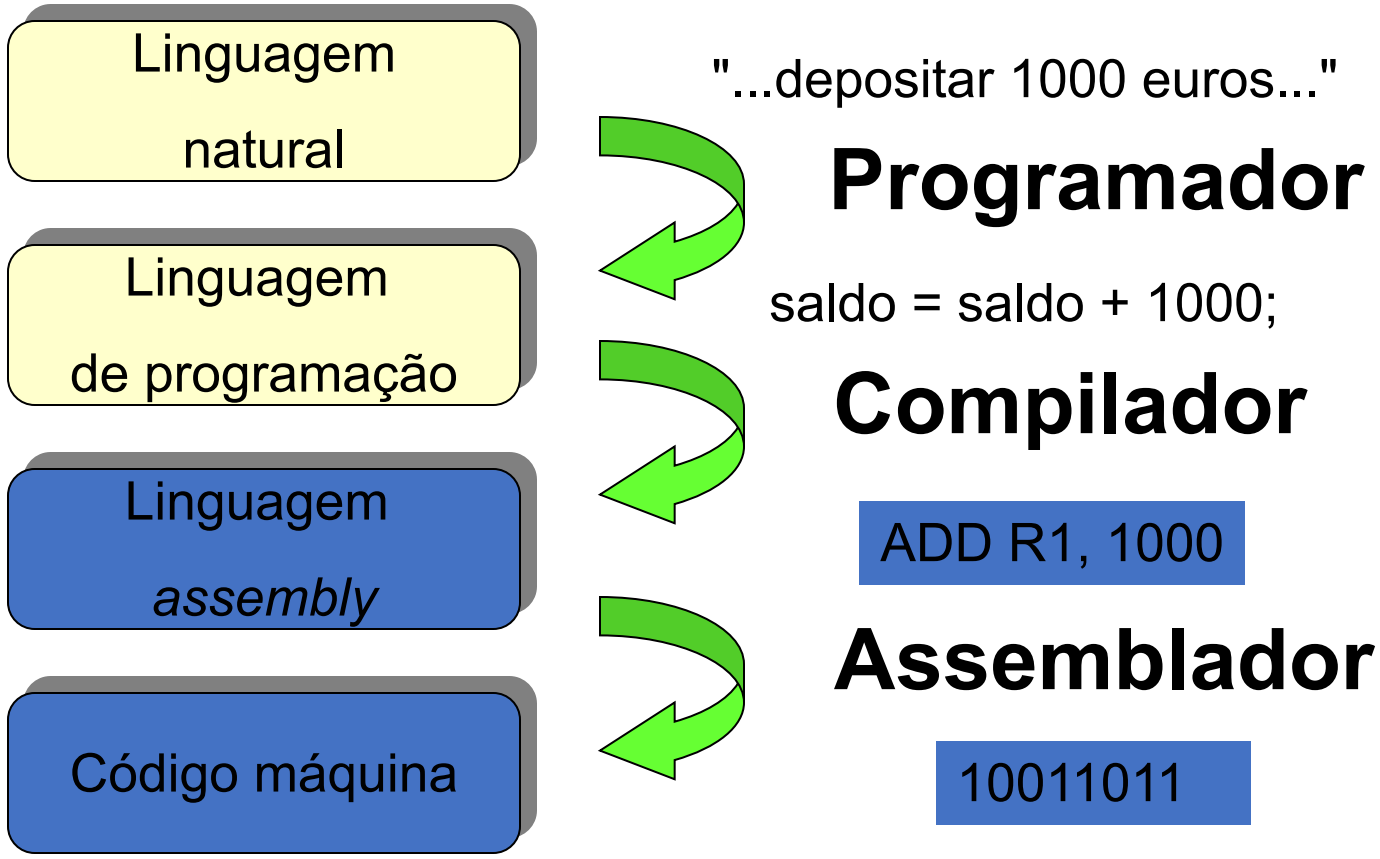
Programação em Assembly



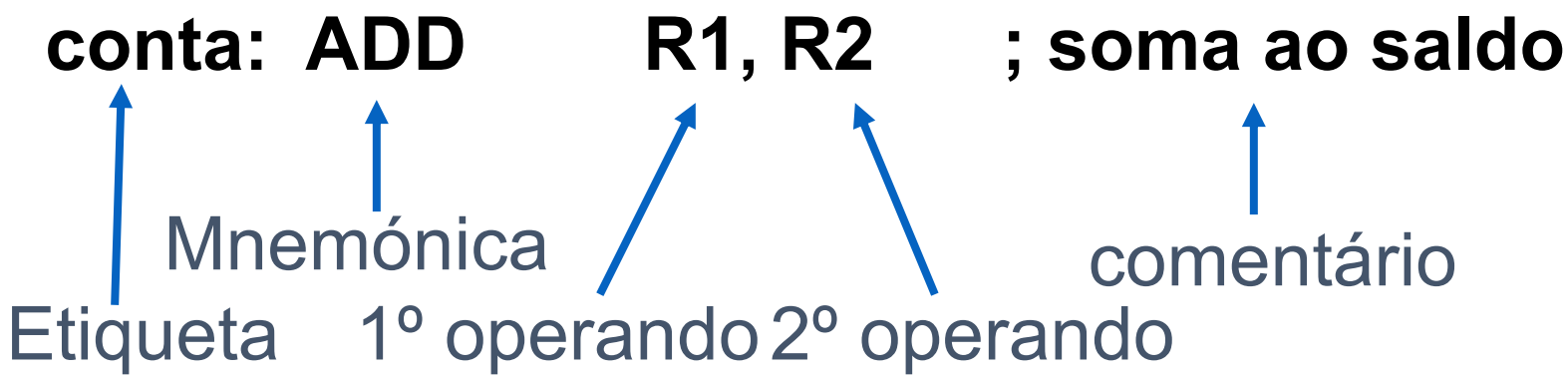
TÉCNICO LISBOA

Programação do computador



Linguagem *assembly*

- Uma instrução por linha
- Formatação rígida
- Comentários *qb* (escritos logo ao fazer o código!)
- As instruções **refletem diretamente** os recursos do processador



Comentários das instruções

- Em cada linha, o **assembler ignora o caráter ";" e os que se lhe seguem** (até ao fim dessa linha)
- Praticamente **todas** as linhas de *assembly* devem ter comentário, pois a programação é de baixo nível.

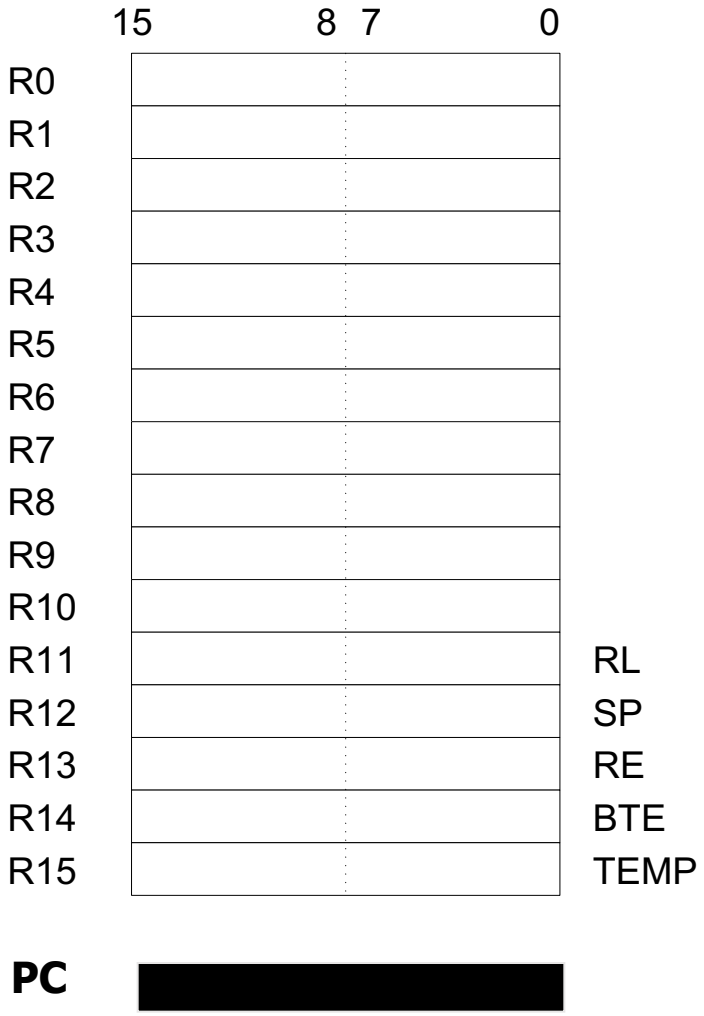
Sem comentários. O que é isto?!

```
                PLACE   1000H           ; Início da zona de dados em 1000H
Saldo:          WORD    0               ; variável Saldo inicializada a 0
                PLACE   0000H           ; Início do programa em 0000H
Deposita:       MOV     R3, 100         ; Valor a depositar na conta
                MOV     R1, Saldo       ; Endereço da variável Saldo em R1
                MOV     R2, [R1]        ; Lê o valor do saldo
                ADD     R2, R3          ; Acrescenta o valor a depositar ao saldo
                MOV     [R1], R2        ; Escreve a variável Saldo
                . . .                   ; . . .
```

TÉCNICO LISBOA

Registos do processador

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



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**
- Exemplo: **flags.asm**

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

Classes de instruções

Classe de instruções	Descrição e exemplos
Instruções **aritméticas**	Lidam com números em complemento para 2 **ADD, SUB, CMP, MUL, DIV**
Instruções **lógicas**	Lidam com sequências de bits **AND, OR, SET**
Instruções de **deslocamento**	Deslocam os bits de um registo **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**

Representação de números

- Os registos têm **16 bits** (4 dígitos hexadecimais)
- Nas instruções aritméticas (ADD, etc.), os valores estão em **complemento para 2** (entre 8000H e 7FFFH)
- Nas instruções de bit (e.g., lógicas, deslocamento), os registos são apenas um conjunto de **bits individuais**
- Nas instruções de **controlo de fluxo** (saltos), os valores dos registos são considerados **endereços, sem sinal** (0000H a FFFFH)
- Exemplo: **representação_números.asm**

Instruções aritméticas, lógicas e de deslocamento

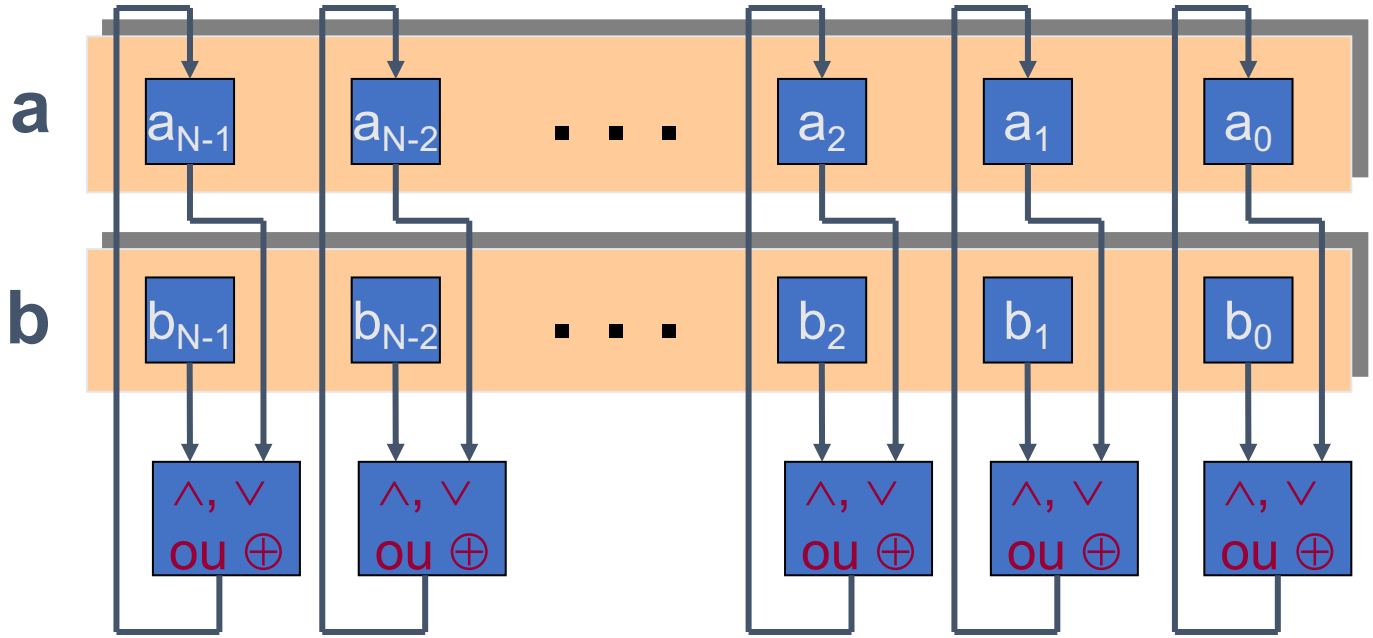
Instruções aritméticas típicas

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

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 lógicas em *assembly*

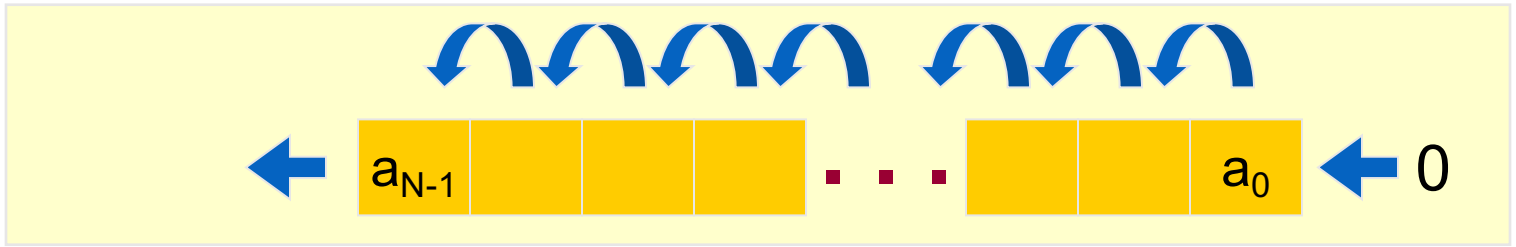
AND	a, b	$a_i \leftarrow a_i \wedge b_i$ ($i \in 0..N-1$)
OR	a, b	$a_i \leftarrow a_i \vee b_i$ ($i \in 0..N-1$)
XOR	a, b	$a_i \leftarrow a_i \oplus b_i$ ($i \in 0..N-1$)
NOT	a	$a_i \leftarrow \overline{a_i}$ ($i \in 0..N-1$)



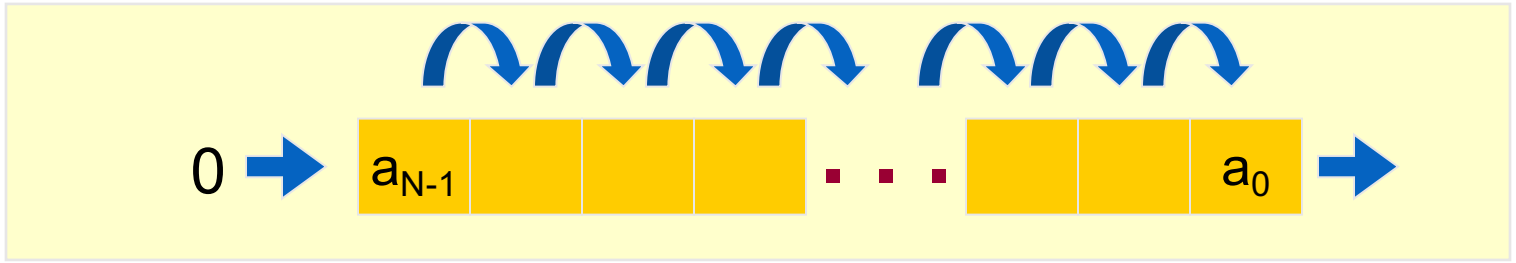
Instruções de deslocamento

- Correspondem a **multiplicar** (SHL) e **dividir** (SHR) por 2^n.
- Exemplo: **instruções_bit.asm**

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

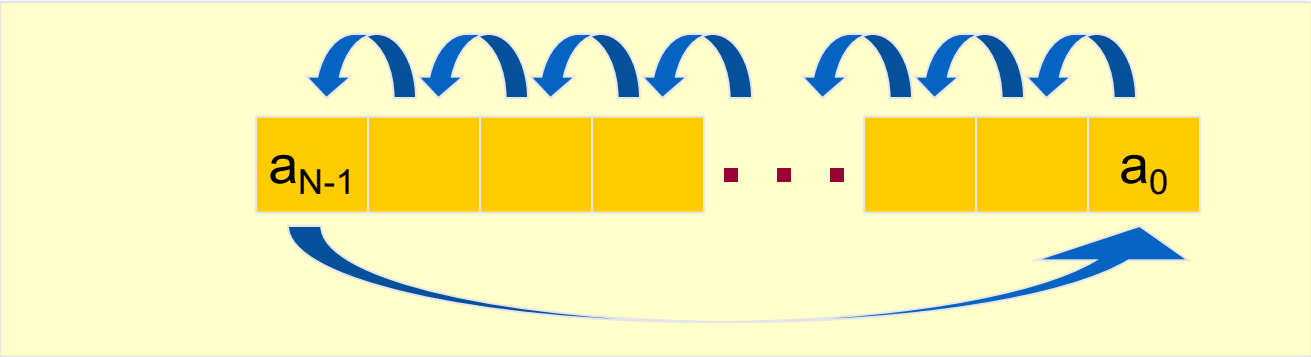


SHR a, n $n * [a_i \leftarrow a_{i+1} \ (i \in 0..N-2); a_{N-1} \leftarrow 0]$

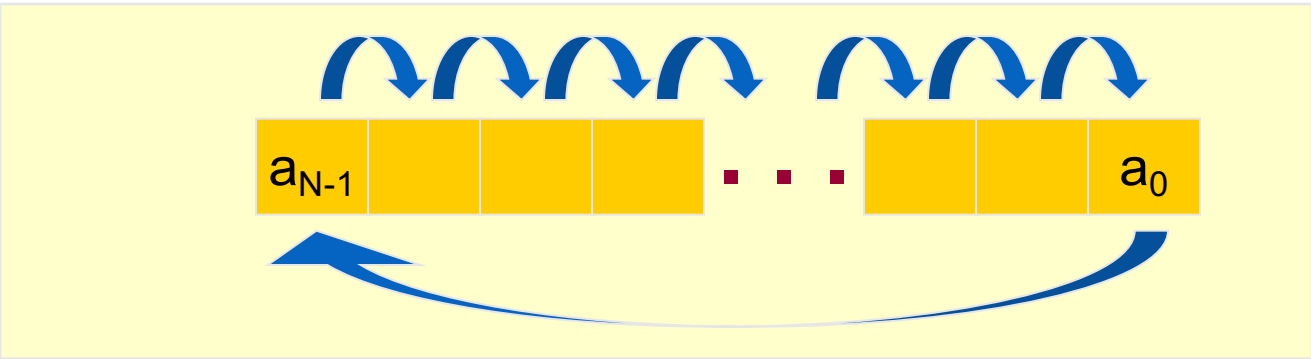


Instruções de rotação

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



Instruções de transferência de dados

Transferência de dados (16 bits)

Acesso	Instruções		Descrição	Comentários
Interno	MOV	Rd, k	Rd ← k	Coloca a constante k em Rd k ∈ [-32768 .. +32767]
	MOV	Rd, Rs	Rd ← Rs	Copia o reg. *Rs* para o reg *Rd*
Memória	MOV	Rd, [Rs] Rd, [k]	Rd ← M[Rs] Rd ← M[k]	Lê 16 bits da memória: endereço num registo ou constante
	MOV	[Rd], Rs [k], Rs	M[Rd] ← Rs M[k] ← Rs	Escreve 16 bits na memória: endereço num registo ou constante

```
MOV     R1, 3          ; carrega o R1 com 3
MOV     R2, R1         ; agora o R2 fica também com 3
MOV     R3, 1000H      ; carrega o R3 com 1000H
MOV     [R3], R2       ; escreve 3 na memória, no endereço 1000H
MOV     [2000H], R2    ; escreve 3 na memória, no endereço 2000H
MOV     R5, [R3]       ; lê a memória (endereço 1000H) para o R5
MOV     R6, [2000H]    ; lê a memória (endereço 2000H) para o R6
```

Constantes podem (e devem) ser simbólicas (EQU)

Acessos à memória em 8 bits

Instruções		Descrição	Comentários
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

```
MOV     R1, 1234H
MOV     R2, 1000H
MOVB    [R3], R2   ; escreve 34H no endereço 1000H (só um byte)
MOVB    R4, [R3]   ; lê o byte no endereço 1000H (e só esse byte)
```

- Com MOVB, o endereço acedido:
  - Tem de ser um registo (não pode ser uma constante)
- Exemplo: **acessos_memória.asm**

Swap

Instruções	Descrição	Comentários
SWAP Rd, [Rs]	TEMP ← M[Rs] M[Rs] ← Rd Rd ← TEMP	TEMP = registo temporário

Instruções de controlo de fluxo

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)

Diretivas

Diretivas (pseudo-instruções)

- São diretivas para o assembler e não instruções para o microprocessador. Logo, **não geram código executável**.

- Pseudo-instruções típicas:
  - EQU
  - PLACE
  - WORD
  - TABLE
  - BYTE

Diretiva EQU

- **Não é uma instrução** (não gera código)
- Serve apenas para definir o **valor de constantes** simbólicas
- Não é uma etiqueta (*label*), pelo que não leva ":"

- Sintaxe:
  símbolo EQU *constante-literal*

- Exemplo:

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

PLACE

- Permite indicar o **endereço a partir do qual** as instruções ou variáveis seguintes devem ficar localizadas
- Até aparecer um PLACE, considera-se que há um **PLACE 0 implícito**, desde o início do programa
- 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
. . .           . . .
```

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 do ficheiro).

```
            PLACE   0100H       ; início dos endereços dos dados

OLA         EQU     4           ; constante definida com o valor 4
VAR1:       WORD    10          ; reserva uma palavra no endereço 0100H
VAR2:       WORD    OLA         ; reserva uma palavra no endereço 0102H

            PLACE   0000H       ; início dos endereços das instruções
inicio:
            MOV     R1, OLA     ; R1 ← 4 (isto é um dado)
            MOV     R2, VAR2    ; R2 ← 0102H (isto é um endereço)
            . . .               ; resto do programa
```

WORD

- Define (**reserva espaço**) uma **variável** de 16 bits (word)
- A mesma diretiva permite definir várias variáveis de uma word consecutivas
- Cada variável gasta **2 bytes** (uma word)

- Sintaxe:

  etiqueta: WORD *constante* {, *constante*}

- Exemplo:

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

WORD é diferente de EQU

```
        PLACE   0100H       ; início dos endereços gerados
                            ; pelo assemblador (zona de dados)
OLA     EQU     4           ; constante definida com o valor 4 (não
                            ; "gasta" endereços do assemblador!)
VAR1:   WORD    10          ; reserva uma palavra de memória, localizada
                            ; no endereço 0100H (valor de VAR1) e
                            ; inicializa-a com 000AH
VAR2:   WORD    OLA         ; Idem, no endereço 0102H (valor de VAR2) e
                            ; inicializa-a com 4 (valor de OLA)

        PLACE   0000H       ; início da zona de código
inicio:                     ; inicio vale 0000H
        MOV     R1, OLA     ; R1 ← 4 (isto é uma constante de dados)
        MOV     R2, VAR2    ; R2 ← 0102H (isto é um endereço)
```

Acesso à memória do WORD

```
        PLACE   0100H       ; início dos endereços
OLA     EQU     4           ; constante definida com o valor 4
VAR1:   WORD    10          ; reserva uma palavra no endereço 0100H
VAR2:   WORD    OLA         ; reserva uma palavra no endereço 0102H

        PLACE   0000H
inicio:
        MOV     R1, OLA     ; R1 ← 4 (isto é um dado)
        MOV     R2, VAR2    ; R2 ← 0102H (isto é um endereço)
                            ; isto NÃO acede à memória!

    ; agora sim, vamos aceder à memória
        MOV     R3, [R2]    ; R3 ← M[VAR2], ou
                            ; R3 ← M[0102H]
                            ; R3 fica com 4 (valor do OLA)
        MOV     R4, 5AH
        MOV     [R2], R4    ; M[VAR2] ← 5AH, ou
                            ; M[0102H] ← 5AH
```

- Exemplo em: **diretiva_word.asm**

TABLE

- Define (**reserva espaço**) uma **tabela com várias variáveis de 16 bits** (words)
- Só reserva espaço (não inicializa)

- Sintaxe:
  etiqueta: TABLE *constante*

- Ocupa 2 * constante bytes

- Exemplo

```
T1:     TABLE   10H  ; reserva espaço para 16 words
                     ; (32 bytes)
                     ; A primeira fica localizada no endereço atribuído a T1, a
                     ; segunda em T1 + 2, etc.
```

TABLE vs tabelas com WORD

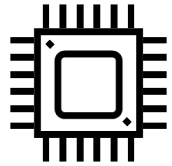
- A TABLE só reserva espaço (**não inicializa**).
  - É boa para reservar uma área que depois se pode ir escrevendo ao longo do programa

- Para definir **tabelas de constantes**, é melhor definir os vários elementos da tabela com WORDs:

```
lista:  WORD    12      ; valores da lista
        WORD    5
        WORD    -3
        WORD    4
        WORD    2
        WORD    -1
        WORD    8
```

- Exemplo: **soma_words.asm**
- Exemplo: **soma_words_indexed.asm**

BYTE



- Define (reserva espaço) uma **variável de 8 bits (byte)**
  - **ATENÇÃO:** para valores negativos, devem usar-se variáveis WORD e não BYTE (os valores negativos num processador precisam sempre dos bits todos)
- A mesma diretiva permite definir várias variáveis de um byte consecutivas
- Sintaxe:

  etiqueta: BYTE *constante* {, *constante*}
- Exemplo (gasta 5 bytes):

```
S1:     BYTE 'a', “ola”, 12H        ; lista de bytes
```

- Exemplo: **conta_bytes.asm**

Modos de endereçamento

Modos de endereçamento

Modo	Exemplo		Comentário
Implícito	CALL	*etiqueta*	Manipula SP implicitamente
Imediato	ADD CMP	R1, 3	Só entre -8 e +7
Registo	ADD	R1, R2	
Direto	MOV	R1, [1000H]	
Indireto	MOV	R1, [R2]	
Baseado	MOV	R1, [R2 + 6]	Constante par, de -16 a + 14
Indexado	MOV	R1, [R2 + R3]	
Relativo	JMP JZ	*etiqueta*	Só dá para aprox. PC $\pm 2^{12}$ Só dá para aprox. PC $\pm 2^{8}$

Exemplos de programas no PEPE

Exemplo de programa no PEPE

- Objetivo do programa: somar um número com todos os inteiros positivos menores que ele.

$$soma = N + (N-1) + (N-2) + \ldots + 2 + 1$$

1.	soma ← 0	(inicializa **soma** com zero)
2.	iteracao ← N	(inicializa **iteracao** com N)
3.	Se (iteracao < 0) salta para 8	(se **iteracao** for negativo, salta para o fim)
4.	Se (iteracao = 0) salta para 8	(se **iteracao** for zero, salta para o fim)
5.	soma ← soma + iteracao	(adiciona **iteracao** a **soma**)
6.	iteracao ← iteracao – 1	(decrementa **iteracao**)
7.	Salta para 4	(salta para o passo 4)
8.	Salta para 8	(fim do programa)

Programa no PEPE

```
        ; Utilização dos registos:
        ; R0 – soma
        ; R1 – iteracao

        N       EQU     5           ; definição do N

1.              MOV     R0, 0       ; soma ← 0
2.              MOV     R1, N       ; iteracao ← N
3.  maisUm:     CMP     R1, 0       ; se (iteracao <= 0) salta para fim
4.              JLE     fim         ; junta os dois testes
5.              ADD     R0, R1      ; soma ← soma + iteracao
6.              SUB     R1, 1       ; iteracao ← iteracao – 1
7.              JMP     maisUm      ; salta para mais uma iteração
8.  fim:        JMP     fim         ; "termina"
```

- Exemplo: **soma_série.asm**

Outro exemplo: contar bits a 1

posição em teste	Máscara	Valor (76H)	Valor AND máscara	Bit a 1	Contador de bits a 1
0	0000 000**1**	0111 011**0**	0000 000**0**	Não	0
1	0000 00**1**0	0111 01**1**0	0000 00**1**0	Sim	1
2	0000 0**1**00	0111 0**1**10	0000 0**1**00	Sim	2
3	0000 **1**000	0111 **0**110	0000 **0**000	Não	2
4	000**1** 0000	011**1** 0110	000**1** 0000	Sim	3
5	00**1**0 0000	01**1**1 0110	00**1**0 0000	Sim	4
6	0**1**00 0000	0**1**11 0110	0**1**00 0000	Sim	5
7	**1**000 0000	**0**111 0110	**0**000 0000	Não	5

1. **contador** ← 0 (inicializa contador de bits a zero)
2. **máscara** ← 01H (inicializa máscara a 0000 0001)
3. Se (**máscara** ∧ **valor** = 0) salta para 5 (se o bit está a zero, passa ao próximo)
4. **contador** ← **contador** + 1 (bit está a 1, incrementa contador)
5. Se (**máscara** ← 80H) salta para 8 (se já testou a última máscara, termina)
6. **máscara** ← **máscara** + **máscara** (duplica máscara para deslocar bit para a esquerda)
7. Salta para 3 (vai testar o novo bit)
8. Salta para 8 (fim do algoritmo)

Programa no PEPE

```
valor           EQU   76H       ; Valor cujo número de bits a 1 é para ser contado
mascaraInicial  EQU   01H       ; 0000 0001 em binário (máscara inicial)
mascaraFinal    EQU   80H       ; 1000 0000 em binário (máscara final)

; Utilização dos registos:
; R0 – auxiliar (valores intermédios)
; R1 – contador de bits a 1
; R2 – máscara

inicio:     MOV   R1, 0             ; Inicializa o contador de bits com zero
            MOV   R2, mascaraInicial ; Inicializa valor da máscara
            MOV   R0, valor         ; Cópia do valor
teste:      AND   R0, R2            ; Isola o bit que se quer ver se é 1
            JZ    proximo           ; Se o bit for zero, passa à máscara seguinte
            ADD   R1, 1             ; O bit é 1, incrementa o valor do contador
proximo:    MOV   R0, mascaraFinal
            CMP   R2, R0            ; Compara com a máscara final
            JZ    fim               ; Se forem iguais, já terminou
            SHL   R2, 1             ; Desloca bit da máscara para a esquerda
            JMP   teste             ; Vai fazer mais um teste com a nova máscara
fim:        JMP   fim               ; Fim do programa
```

- Exemplo: **conta_uns.asm**

Contar bits sem mascaras: só com deslocamentos

posição em teste	Valor (76H)	Bit a 1	Contador de bits a 1
0	0111 011**0**	Não	0
1	0011 101**1**	Sim	1
2	0001 110**1**	Sim	2
3	0000 111**0**	Não	2
4	0000 011**1**	Sim	3
5	0000 001**1**	Sim	4
6	0000 000**1**	Sim	5
7	0000 000**0**	Não	5

1. **contador** ← 0 (inicializa contador de bits a zero)
2. Se (**valor** = 0) salta para 7 (se o valor é zero, acabou)
3. **valor** ← **valor** >> 1 (desloca o valor de um bit para a direita, bit fica na flag C)
4. Se (**flag C = 0**) salta para 2 (bit era 0, não incrementa contador)
5. **contador** ← **contador** + 1 (bit era 1, incrementa contador)
6. Salta para 2 (vai testar o novo bit mais à direita)
7. Salta para 7 (fim do algoritmo)

Agora, contagem de bits a 1 com deslocamentos (shifts)

```
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  (afinal não se perde logo)
            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
```

Bibliografia

Recomendada

- **[Delgado&Ribeiro_2014]**
  - **Secções 3.1-3.5**

Secundária/adicional

- [Patterson&Hennessy_2021]
  - Cap. 2