Introdução ao Assembler FASM – Flat Assembler (*versão preliminar*)

1 Introdução

Este texto constitui uma pequena introdução ao *assembler* FASM(Flat Assembler). Embora contenha algumas notas relacionadas com a arquitectura de computadores e programação em baixo-nível, não é um manual de programação. Pretende ser apenas um auxiliar para apoiar os primeiros passos na programação em *assembly* usando o *assembler* FASM.

O primeiro conceito que importa esclarecer é precisamente a distinção entre aqueles dois termos realçados a itálico. Assim, o termo *assembly* refere-se à linguagem de programação, que é também designada por linguagem de baixo-nível, uma vez que se encontra intimamente relacionada com o processador a que se destina. Deste modo, cada processador, de cada fabricante(Intel, AMD, Motorola,...), tem o seu próprio *assembly*, já que cada um tem estrutura interna diferente, mas o termo *assembly* aplica-se a todos eles (i.e. não há uma linguagem "assembly Intel" ou "assembly AMD", tal como existe Pascal ou C). O que acontece é dizer-se que se está a utilizar o *assembly* do Pentium, do Athlon, ou do Motorola68000. Em princípio, um programa que utilize o *assembly* do Pentium não será executado por um processador de outro fabricante, a menos que sejam compatíveis entre si.

Pelo seu lado, o termo *assembler* ("montador" em inglês) refere-se a um programa que permite facilitar o trabalho com a linguagem *assembly*, fazendo com que esta se assemelhe um pouco mais a uma linguagem de alto-nível. De facto, torna-se muito complicado para os programadores humanos escrever programas usando a linguagem "pura" do processador (linguagem-máquina), a qual é constituída por um conjunto mais ou menos extenso de bits (ex: a instrução "mov ah,40h", muito usada em assembly, corresponde a 1011010010000000, de facto bits é a única coisa que as máquinas "entedem" !). O *assembler* atribui nomes (mnemónicas) aos conjuntos de bits que constituem as instruções do processador, facilitando a sua compreensão pelos humanos. O *assembler* também chama a si a execução de um conjunto de acções necessárias para que um programa possa ser executado (p.ex. o controlo de certos registos do processador), escondendo essas operações ao programador.

A programação em *assembly* apresenta algumas características próprias. A primeira é que permite escrever programas que executam muito mais rapidamente que programas escritos em linguagens de alto-nível. Isto deve-se ao facto de que os compiladores ou interpretadores destas linguagens, ao traduzirem as suas instruções para *assembly*, o fazerem de forma pouco eficiente, gerando mais instruções *assembly* do que um programador humano pode conseguir se programar directamente em baixo-nível. Para tirar partido deste facto, quase todas as linguagens de alto-nível permitem que se possam embutir instruções *assembly* entre as instruções da própria linguagem, precisamente naqueles sítios em que for detectado que a execução do programa está a gastar mais tempo. Os programas escritos em *assembly* ficam assim mais pequenos e logo mais rápidos. Uma outra característica do *assembly* é o controle que proporciona sobre os componentes de hardware, em particular do processador, permitindo usar todas as suas funcionalidades e capacidades. Importa notar que certas linguagens de alto-nível impedem ou limitam o acesso a certos componentes de hardware, com a

finalidade de evitar que possam ser desencadeadas acções potencialmente perigosas. Porém, em algumas situações, pode ser necessário o acesso a essas funcionalidades do hardware, o que pode ser conseguido através do *assembly*. Entretanto, uma vez que neste caso não existe um compilador para controlar as acções do programador, ficam por sua conta e risco as consequências dessas acções.

Existem vários assemblers, entre os quais os mais famosos são o MASM(Microsoft) e o TASM(Borland), que são propriedade dos respectivos fabricantes e logo são pagos. Entre os exemplos gratuitos temos o NASM(http://sourceforge.net/projects/nasm) e o FASM que pode ser descarregado de http://flatassembler.net/, onde para além do próprio programa pode ser encontrada literatura de apoio, exemplos, utilitários, etc. De um modo geral, o FASM é mais fácil de utilizar do que outros programas similares. Nas alíneas seguintes apresentam-se algumas das suas principais características.

Antes de passar à análise dos aspectos mais relevantes relacionados com a escrita de programas em assembly usando o FASM, convém rever alguns conceitos referentes ao tratamento da informação por parte dos computadores digitais.

2 Codificação da informação

Os computadores digitais usam bits para representar a mais pequena quantidade de informação. De facto, um *bit* (binary digit) assume apenas um de dois estados possíveis designados por *false* e *true*, habitualmente representados respectivamente por "F" e "T" ou "0" e "1" (neste texto será utilizada a segunda terminologia). Esta codificação diz-se binária precisamente porque existem apenas dois estados possíveis para um bit. Em termos de implementação física esses dois estados são traduzidos pela não existência de uma grandeza eléctrica como corrente ou tensão para o caso do "0" e pela existência de um certo valor para essa grandeza, habitualmente o valor de tensão de 5V, para o caso do "1". Se com um bit podem ser representados dois estados (0,1), então com dois bits podem representar-se quatro estados (00,01,10,11), três bits permitem oito casos, etc; de cada vez que se acrescenta um bit duplicam os casos. A regra é que com *n* bits podem codificar-se $m = 2^n$ casos.

2.1 Representação de valores numéricos

A expressão da significância posicional constante do AnexoC, indica o modo como os bits podem ser usados para representar valores numéricos usando apenas os algarismos binários 0 e 1, tal como as pessoas o fazem usando os algarismos decimais de 0 a 9.

Os números são frequentemente representados noutras bases para além da binária, para simplificar o seu tratamento. Como a informação contida num bit é pequena (apenas 0 ou 1) torna-se necessário trabalhar com um grande número deles para representar informação realmente útil. Assim, recorre-se ao agrupamento dos bit em unidades maiores (byte, Kbyte,...) ou ainda à representação dos valores noutras bases de numeração, o que corresponde ao agrupamento dos bit em unidades maiores (p. ex. cada algarismo hexadecimal é formado por quatro algarismos binários). A partir daqui define-se uma aritmética binária, que permite realizar as operações aritméticas usando dados binários, tal como na aritmética decimal.

2.2 Representação de símbolos alfanuméricos

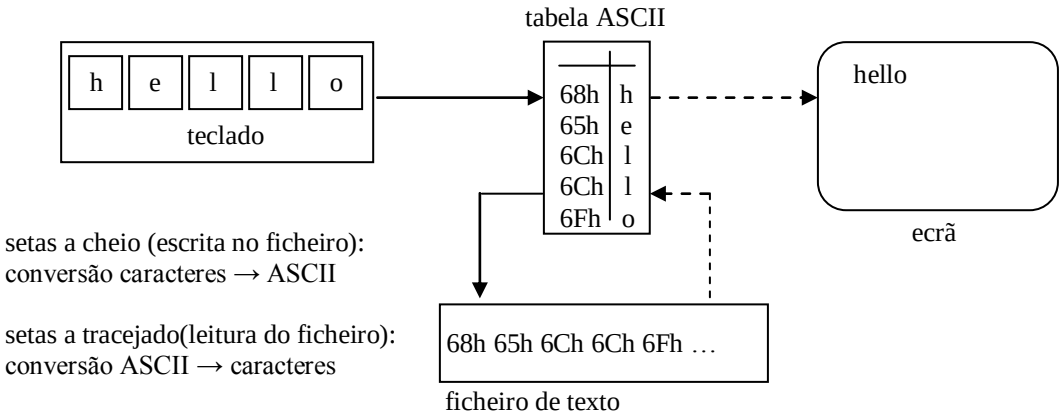
A expressão da significância posicional indica o modo como valores numéricos decimais podem ser representados em binário e vice-versa; trata-se de uma relação matemática entre números, bem definida e que não deixa ambiguidades na representação. Maiores dificuldades surgem na representação em binário de grandezas não numéricas como, por exemplo, os caracteres alfabéticos ou os sinais de pontuação. Note-se que a necessidade de traduzir estes valores para binário decorre do facto de que os *bits*(zeros e uns) são o único tipo de informação que os computadores digitais podem armazenar e tratar. Ora não é fácil estabelecer uma relação entre símbolos alfanuméricos e valores numéricos, a menos que essa relação seja estabelecida de alguma forma arbitrária. Por exemplo, ao carácter 'A' pode ser atribuído o valor 00 ou 111 ou qualquer outro, sem que algum deles esteja mais correcto que os outros. Deste modo surgiram várias propostas de tabelas que relacionam os caracteres alfanuméricos com valores binários, entre elas a tabela ASCII (American Standard Code for Information Interchange). Esta tabela, que se encontra no AnexoA, indica o valor binário correspondente a cada um dos caracteres alfanuméricos que o computador pode tratar (a versão original usava 7 bits, permitindo codificar 2^7=128 símbolos entre letras, dígitos e sinais de pontuação, sem acentos; posteriormente foi expandida para 8 bits codificando 2^8=256 símbolos, de modo a contemplar os acentos e outros caracteres especiais). Outro exemplo é a tabela UNICODE a qual utiliza 16 bits para representar de forma internacional e única qualquer símbolo, facilitando a escrita do software (os primeiros 256 caracteres são iguais aos da tabela ASCII). O problema do UNICODE é que só suporta 64K (2^{16}=65536) símbolos, mas há mais de 200 000 símbolos em todas as línguas do mundo; embora não seja ainda universalmente usado, o Windows e Java já o usam como standard.

Seja qual for a tabela utilizada, os códigos que a compõem são os utilizados sempre que um carácter alfanumérico é recebido do teclado para ser armazenado num ficheiro, ou é enviado para o ecrã ou para uma impressora.

2.3 Armazenamento da informação [1]

Relativamente à representação de valores numéricos e de símbolos alfanuméricos, convém tecer algumas considerações. Para tal, vamos considerar que é usada a tabela ASCII com códigos em hexadecimal, uma vez que é a tabela mais utilizada (ver Anexo A, pág. 14). Quando se introduz um texto pelo teclado, por exemplo a palavra "hello", cada uma das teclas ao ser pressionada gera o correspondente código ASCII. Por exemplo, a tecla "h" gera o código 68h, a tecla "e" o código 65h, etc. Deste modo, ao escrever aquela palavra, o teclado envia a sequência 68h+65h+6Ch+6Ch+6Fh. Se esta palavra for gravada num ficheiro de texto, estes valores (que são números!) são aqueles que ficam armazenados. Ao ler o ficheiro, a máquina recupera aqueles valores (que são números, recorde-se) e converte-os para os respectivos símbolos, por consulta da tabela ASCII (fazendo a conversão ao contrário). Deste modo o que aparece no ecrã é de novo a palavra "hello" (ver o diagrama a seguir).

[1] O AnexoB descreve o modo como estes conceitos podem ser comprovados recorrendo ao utilitário DEBUG



Situação diferente ocorre quando o valor introduzido pelo teclado corresponde a um valor numérico, como um inteiro. Tomando como exemplo o valor 741 (valor em base decimal), tal como no caso anterior cada tecla gera o código ASCII que lhe corresponde, ou seja 37h+34h+31h. Se este valor for armazenado num ficheiro de texto tudo se passa como anteriormente. Ao ler o ficheiro aqueles códigos são de novo convertidos para os caracteres ASCII e no ecrã aparecerá 741. O primeiro aspecto que importa entender é que o que aparece no ecrã é sempre texto, portanto 741 é por assim dizer a palavra "741" (tal como no caso anterior a palavra era "hello"), ou, dito de outro modo, trata-se do símbolo "7", seguido do símbolo "4", seguido do símbolo "1". Cada um daqueles símbolos não tem qualquer relação com os outros, apenas estão dispostos em sequência; parece um número apenas porque estamos habituados a interpretar como números as sequências de símbolos formadas por algarismos.

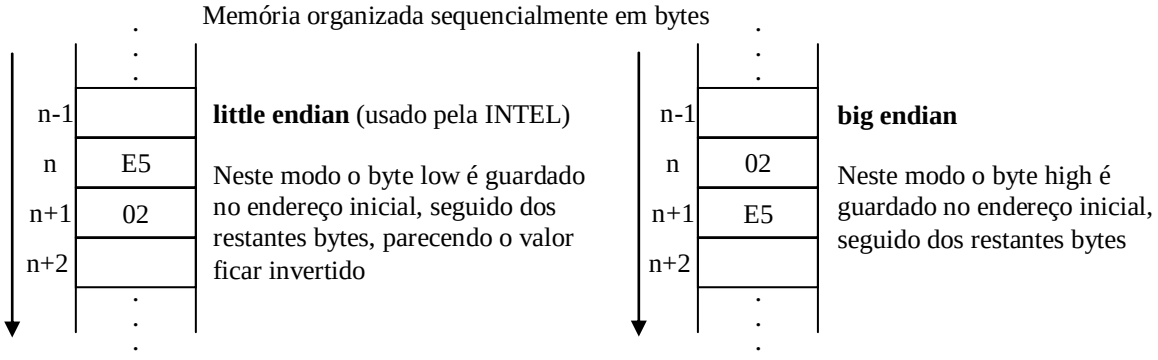
Mas, e se este valor deve ser tratado como um número inteiro? Por exemplo se for necessário operar sobre ele, como multiplicá-lo por outro? É claro que não podem aplicar-se operações aritméticas sobre sequências de símbolos, é preciso fazer com que essas sequências sejam números.

No exemplo em estudo é preciso fazer com que a sequência 741 deixe de ser um "7" encostado a um "4" por sua vez encostado a um "1" e passe a ser $7*10^2+4*10^1+1*10^0=741$.

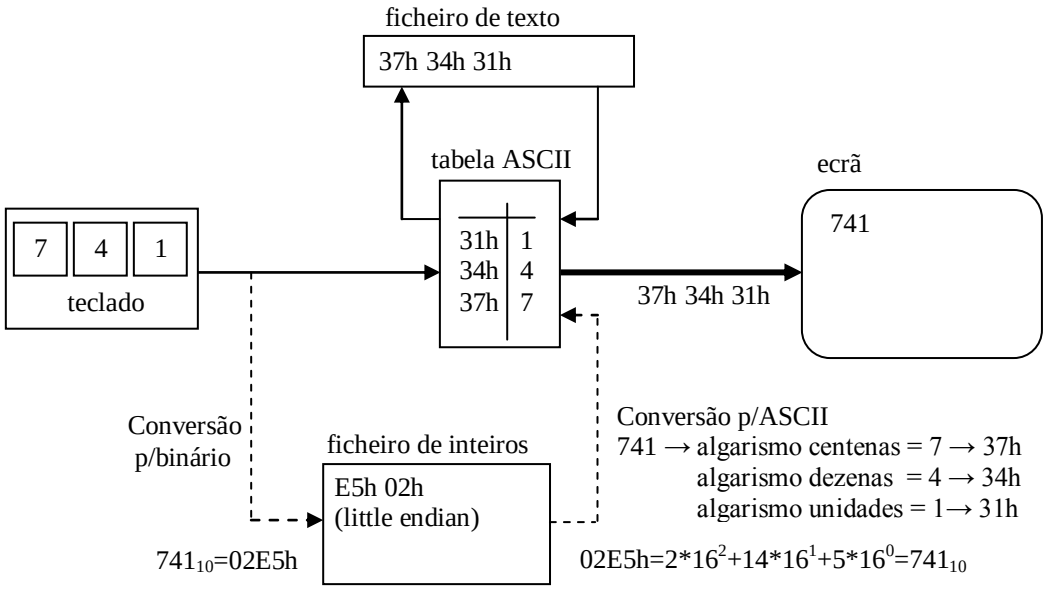
O que deverá então fazer um programa ao ler um valor para uma variável do tipo inteiro? Enquanto não se carrega na tecla "Enter" o buffer de teclado vai armazenando os códigos ASCII das teclas que vão sendo introduzidas (37h+34h+31h no exemplo). A partir do momento em que se carrega no Enter indica-se à máquina que a introdução do valor terminou e que a sequência de códigos deve ser convertida para um inteiro. A primeira operação é obter o valor numérico de cada algarismo a partir do respectivo código ASCII. Ora como o código ASCII dos algarismos é dado pela soma de 30h com o próprio algarismo, basta retirar este 30h a cada código lido, para obter o valor numérico. No exemplo em estudo isso corresponde a fazer: 37h-30h=7, 34h-30h=4, 31h-30h=1. Como a máquina "sabe" que foram introduzidas três teclas, torna-se fácil aplicar a fórmula $7*10^2+4*10^1+1*10^0$ de modo a obter 741. Note-se que este 741 é agora um valor numérico e não meramente três caracteres alfanuméricos seguidos. Sendo assim, e para que possa sofrer operações aritméticas as quais são necessariamente

efectuadas em binário, deve ser aplicada a conversão para base 2, de que resulta: 741_{10}=02E5h=0000001011100101b (acrescentaram-se zeros à esquerda para obter dois bytes completos). O valor 741 é assim constituído pelos dois byte 02h e E5h, em que o primeiro tem mais peso (ou é mais significativo) que o segundo, sendo por isso designados por byte mais significativo (*high byte*) e byte menos significativo (*low byte*).

Finalmente, que acontece se este valor tiver de ser guardado num ficheiro de inteiros? Como a memória é um conjunto de bytes organizados sequencialmente, de que modo são guardados os dois byte do valor? A figura seguinte ilustra os dois modos possíveis, designados por *little endian* e *big endian* [2]. Os processadores da INTEL usam o modo *little endian*.



O diagrama a seguir esquematiza o que acontece quando a sequência de teclas "741" é tratada como uma cadeia de caracteres (texto) ou como um valor numérico (inteiro).



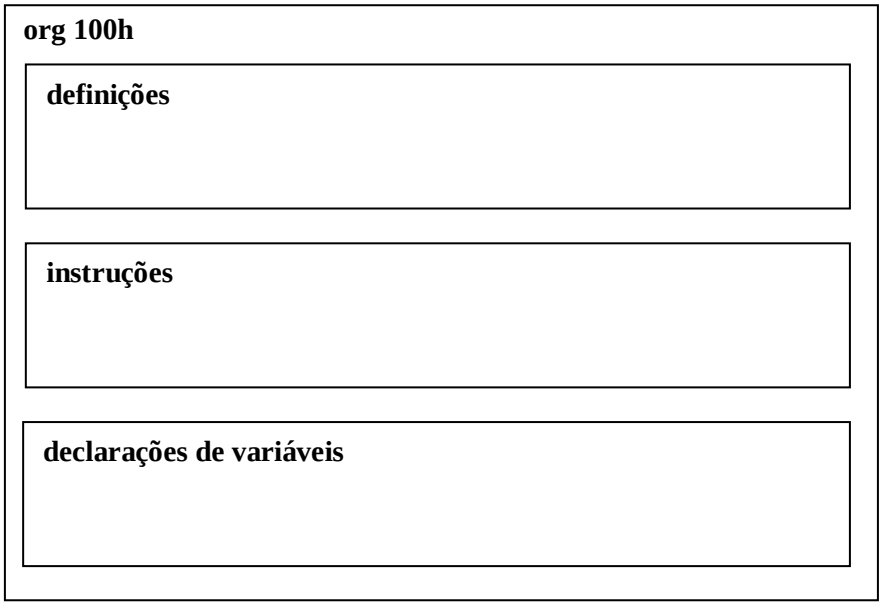
setas a cheio: tratamento de texto (cadeias de caracteres)
setas a tracejado: tratamento de inteiros (valores numéricos)

[2] O termo *endian* tem origem no livro "As viagens de Gulliver" e refere-se à questão de qual dos lados os ovos devem ser quebrados.

3 Estrutura de um programa FASM

Os programas executáveis gerados pelo FASM são do tipo ".COM", ficheiros em binário puro, os quais devem ser carregados e executados a partir do endereço 100h. Para esse efeito, todos os programas devem começar com a directiva **org 100h**, segundo o esquema abaixo:

ficheiro fonte: teste.asm



5 Características principais do FASM

5.1 Comentários: começam por ";" – tudo o que lhe seguir é ignorado pelo *assembler*. Podem ser aplicados a uma linha inteira ou apenas a parte. É boa ideia comentar as partes do programa cujo significado seja menos evidente (principalmente ao fim de algum tempo), como sejam os algoritmos utilizados, significado das variáveis, etc.

5.2 Maiúsculas/minúsculas: o FASM é case-sensitive, ou seja, distingue entre elementos escritos em maiúsculas ou minúsculas. Esta aplica-se a nomes (constantes, variáveis), mas não a instruções, directivas ou comentários. Por exemplo, a instrução "mov" pode ser escrita "Mov" ou "MOV", pois é uma palavra reservada da linguagem, mas uma variável que tenha sido declarada com o nome "foo" não pode ser referenciada por "FOO", caso contrário o FASM indica que a variável não existe.

5.3 Linha de código: uma linha típica de código em FASM tem a forma:

<label>: <instrução> , <operandos> ;comentário ← em que todos os elementos são opcionais
i.e., uma linha de código pode não conter alguns deste elementos

<label>: → indica um local para onde uma instrução de salto pode saltar

(NOTA: a utilização dos dois pontos é opcional, mas é mais seguro utilizá-los. Se não forem usados o FASM pode considerar que uma instrução que foi escrita por engano é uma label , não indicando erro, mas fazendo com que o programa não trabalhe correctamente (ex. se numa linha com uma única

instrução, escrever por engano "lodab" em vez da instrução correcta "lodsb", o FASM considera "lodab" uma label e não dá qualquer erro)

<instrução> → uma das instruções (mnemónicas) do FASM (ex: mov, add, jmp)

<operandos> → constantes, variáveis, registos, etc, a que a instrução faz referência

Exemplo:

```
        mov al , 5    ;coloca o valor inteiro 5 no registo al
cont:   dec al        ;define a label "cont" e decrementa de uma unidade o valor do registo al
        jnz  cont     ;salta (jump) para a linha anterior (aonde foi definida a label "cont") se não
                      ;resultou zero da última operação aritmética realizada (que foi "dec al")
```

P. Qual a acção realizada por este pequeno programa ?
R. Decrementa o registo al desde o valor inicial 5, até atingir 0

5.4 Pseudo-instruções: não correspondem a instruções verdadeiras do processador, mas permitem simplificar certas tarefas como definir constantes ou variáveis. Algumas delas são[3]:

- declaração de dados inicializados: declara dados com valor inicial
  - DB-define byte → os valores são considerados bytes (8 bits)
    db 0x55 (define o byte 0x55) , db 'a' (carácter 'a') , db 0 (inteiro 0) ,
    db 255 (inteiro 255 – maior valor representado por um byte),
    db 'hello',13,10 = db 'h','e','l','l','o',13,10 (define a string 'hello' seguida de CR/LF)
  - DW-define word → os valores são considerados words (16 bits)
    dw 0x1234 (define uma word constituída pelos bytes 0x34 0x12)
    dw 'a' (word 0x41 0x00) , dw 'ab' (word 0x41 0x42)
    dw 65535 (inteiro 65535 – maior valor representado por uma word)
  - DD-define double word → os valores são considerados double-word (32 bits)
    dd 0x12345678 (define uma double-word constituída pelos bytes 0x78 0x56 0x34 0x12)
    dd 1.234567e20 (definição de uma constante em vírgula flutuante)

[3] O FASM dispõe de várias outras; apresentam-se aqui apenas as que vão ser usadas nas aulas práticas

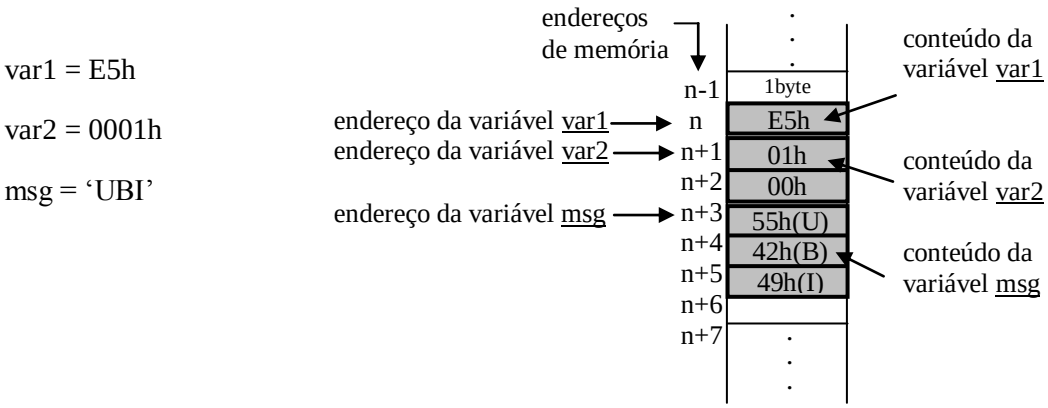
- declaração de dados não inicializados: reserva espaço para armazenar valores
    - RESB-reserve byte → buffer: resb 64 (reserva espaço para 64 bytes)
    - RESW-reserve word → wordvar: resw 1 (reserva espaço para uma word)
    - RESD-reserve double word → doublewordvar: resd 10 (array de 10 double-word)
    - comando EQU: atribui um valor a um símbolo (define uma constante)

      Ex: ecran EQU 1 ;define a constante "ecran" como sendo equivalente a 1

5.5 Referência a conteúdo/endereço de variáveis/memória

NOTA: o nome de uma variável representa o endereço de memória que foi atribuído a essa variável, mais propriamente o endereço do byte inicial dessa variável.

Ex: variáveis var1 tipo *byte*, var2 do tipo *word* (2 byte) e msg do tipo cadeia de caracteres:



- **referências ao conteúdo de uma variável ou posição de memória, exigem que o endereço correspondente seja colocado entre parêntesis rectos "[ ]"** ;

  ex: mov ax, [ind1] ;move o conteúdo da variável ind1 para o registo ax (ax←02E5h)

- **referências ao endereço das variáveis (i.e., à sua posição na memória) não levam parêntesis**

  ex: mov dx, ind2 ;move o endereço da variável ind2 para o registo dx (dx←n+3)

- NOTA: não são permitidas referências à memória/variáveis para origem e destino de dados dentro da mesma instrução. Ex: mov [ind2] , [ind1] → ERRO: não é possível mover o conteúdo de uma variável(memória) directamente para outra variável(memória); o que deverá fazer-se é:

  mov ax , [ind1] ; usa-se um registo auxiliar (neste caso o ax), para

  mov [ind2] , ax ;permitir a operação

- endereços efectivos: qualquer operando de uma instrução que faz referência à memória.

  Exs:

  mov al, [msg] ;coloca no registo al o 1º byte do conteúdo da variável msg (al ← 55h='U')

  mov ah, [msg+1] ;coloca no registo ah o 2º byte do conteúdo da variável msg (ah ← 42h='B')

  mov bl, [msg+2] ;coloca no registo ah o 3º byte do conteúdo da variável msg (bl ← 49h='I')

Outros exemplos:

```
mov  si, msg    ;coloca no registo si, o endereço da variável msg (si ← n+6)
mov  al, [si]   ;coloca no registo al, o conteúdo da posição de memória apontada pelo registo si,
                ;ou seja, o carácter 'U' (al ← 55h='U')
inc si          ;incrementa de uma unidade o registo si (si ← n+6+1)
mov  al, [si]   ;coloca no registo al, o conteúdo da posição de memória apontada pelo registo si,
                ;ou seja o carácter 'B' (ah ← 42h='B')
inc si          ;incrementa de uma unidade o registo si (si ← n+6+1+1)
mov  al, [si]   ;coloca no registo al, o conteúdo da posição de memória apontada pelo registo si,
                ;ou seja o carácter 'I' (bl ← 49h='I')
```

5.6 O FASM não memoriza os tipos das variáveis: quando se declaram variáveis usando as pseudo-instruções para dados inicializados ou não-inicializados, o FASM apenas memoriza o endereço de memória que foi atribuído à variável (para lhe poder aceder), mas "esquece" imediatamente o tipo dessa variável. Isto implica que o **FASM obriga a que se indique o tipo de uma variável sempre que esta é referida**.

Ex1:

```
bytevar:  resb 1          ;declara a variável "bytevar" como sendo um byte (8 bit)
mov [bytevar],10          ;provoca erro, pois o FASM esqueceu o tipo de "bytevar", não conseguindo
                          ;atribuir-lhe o valor 10
mov byte [bytevar],10     ;assim o FASM já consegue atribuir o valor à variável
```

Ex2:

```
wordvar:  resw 1          ;declara a variável "wordvar" como sendo uma word (16 bit)
mov [wordvar],100         ;provoca erro, pois o FASM esqueceu o tipo de "wordvar", não
                          ;conseguindo atribuir-lhe o valor 100
mov word [wordvar],100    ;assim o FASM já consegue atribuir o valor à variável
```

Ex3: quando as expressões envolvem registos não é preciso indicar tipos

```
mov al,10     ;não há erro, pois o FASM "sabe" que o tipo do registo al é byte
mov cx,100    ;não há erro, pois o FASM "sabe" que o tipo do registo cx é word
```

6 Tipos de dados

O FASM reconhece quatro tipos de dados: 1)Number(número), 2)Character(carácter), 3)String(cadeia de caracteres) e 4)Vírgula-flutuante(reais)

1)Number: o FASM usa a numeração decimal por defeito, ou seja, quando se escreve um número ele interpreta-o como estando em decimal; são ainda possíveis a notação hexadecimal, octal e binária.

Exs:

Decimal → 143 = 143_{10} = $1*10^2+1*10^1+1*10^0$ → notação por defeito

Hexadecimal → 013Ch = $1*16^2+1*16^1+12*16^0$ = 284 → valores hexadecimais terminam em "h"

(os valores devem começar sempre por dígitos)

→ 0x13C – outro modo de representar valores em hexadecimal

Octal → 765q=$7*8^2+6*8^1+5*8^0$=501 → valores octais terminam em "q"

Binário → 1001b = $1*2^3+0*2^2+0*2^1+1*2^0$ = 9 → valores binários terminam em "b"

2)Character: uma constante deste tipo consiste num máximo de quatro caracteres entre plicas ou aspas (se forem usadas as plicas dentro da constante poderão aparecer as aspas e vice-versa)

Ex: ‘ab’ , “abcd” , ‘”xy”’, “’yx’”

3)String: só são possíveis de usar com as pseudo-instruções DB, DW, DD. Uma constante do tipo string é semelhante a uma do tipo character, apenas é maior.

Ex:

msg db ‘Ola mundo’ ⇔ db ‘Ola’ , ’ ‘ , ’mundo’ - string

msg1 db ’Bola’ ⇔ db ’B’,’o’,’l’,’a’ - é uma string devido a pertencer à pseudo-instrução “db”, apesar de que tendo só quatro caracteres poderia ser considerada character

4)Floating-point(reais): só possíveis com a pseudo-instrução DD.

Apresentam-se no formato: <digitos> . [<digitos>] [E <expoente>] – o ponto decimal é obrigatório para que o FASM possa distinguir entre inteiros e reais; [ ] significa que é opcional.

```
Exs: dd 1.2                    ;1.2
     dd 1.3e2 ⇔ dd 1.3e+2      ;130.0
     dd 14.e-1                 ;1.4
     dd 3.14                   ;pi
```

7 Exemplo de programa: escrever no ecrã a string "Ola mundo"

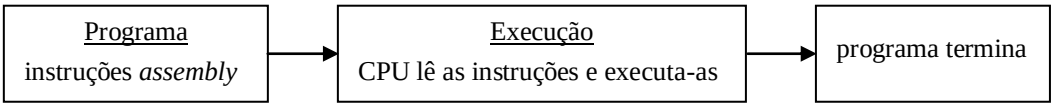
```
org 100h

  mov ah, 40h    ;ah ← 40h (função de escrita)
  mov bx, 1      ;bx ← 1  (1=ecrã)
  mov cx, 9      ;cx ← 9 (número de caracteres a escrever )
  mov dx, msg    ;dx ← endereço da variável "msg" (dx aponta para os dados a escrever)
  int  21h       ;provoca a execução da acção (escrita)

  mov ah, 4Ch    ;ah ← 4Ch (função para terminar a execução de um programa)
  int  21h       ;provoca a execução da acção (termina o programa)

section .data
  msg db “Ola mundo”  ;define a variável "msg"
```

Observe a estrutura e a legibilidade do programa acima; compare com o seguinte:

```
    ORG 100h
   MOV ah, 40h   ;ah ← 40h (FUNÇÃO DE ESCRITA)
  Mov BX, 1   ;bx ← 1  (1=ecrã)
    MOV cx, 9  ;cx ← 11 (número de caracteres a escrever )
   mOV DX, MSG  ;dx ← endereço da variável "msg" (dx aponta para os dados a escrever)
     INT  21h  ;PROVOCA a execução da acção (escrita)
 moV AH, 4Ch  ;ah ← 4Ch (função para terminar a execução de um programa)
  INT  21H  ;provoca a EXECUÇÃO da acção (termina o programa)
SECTion .DATA
msg DB “Ola mundo”  ;DEFINE A VARIÁVEL "msg"
```

8 Programação em Assembly

Como em outras linguagens, programar em *assembly* é escrever uma lista de instruções que o processador vai executar sequencialmente, pela ordem em que foram escritas (embora possa nem sempre ser assim). O exemplo anterior mostra a estrutura típica de um programa *assembly*, o qual pode esquematizar-se da seguinte maneira:



Programa (ver exemplo da pág. anterior)

[directiva org 100h → obrigatória no início de todos os programas]

1) atribuição de valores apropriados aos registos do processador (ver pág. 19), de acordo com a função pretendida - estas funções estão contidas no sistema operativo e são chamadas através de interrupts encontrando-se tabeladas a partir da pág. 20. A tabela contém o código da função e uma breve descrição e ainda os valores de entrada e os registos aonde devem ser colocados bem como os valores de saída que a função devolve nos registos do processador.

ex: função = escrever → int 21h, função 40h (pág.21)

valores a colocar nos registos de entrada

```
mov ah, 40h    ;ah ← 40h (função de escrita)
mov bx, 1      ;bx ← 1  (1=ecrã)
mov cx, 9      ;cx ← 9 (número de caracteres a escrever )
mov dx, msg    ;dx ← endereço da variável "msg" (dx aponta para os dados a escrever)
```

2) chamada ao interrupt - note-se que a atribuição de valores aos registos de entrada só por si não provoca a execução da acção, sendo necessário executar o interrupt correspondente à acção pretendida

cont. do exemplo anterior

int 21h ⇒ ao chegar a esta instrução (interrupt) o processador vai verificar os valores contidos nos registos (que foram lá previamente colocados) e então executa a acção correspondente, neste caso uma acção de escrita no ecrã; se esta instrução não for colocada no programa, o processador não fará acção alguma, mesmo que os valores dos registos de entrada estejam correctamente atribuídos.

3) de modo a terminar correctamente a execução dos programas e o CPU poder continuar com as suas tarefas, todos os programas devem terminar com a sequência (caso contrário o PC pode bloquear):

```
mov ah, 4Ch    ;ah ← 4Ch (função para terminar a execução de um programa)
int 21h        ;provoca a execução da acção (termina o programa)
```

NOTA: atenção à escrita das instuções, em particular das chamadas aos interrupts

ex: int 21h → contém pelo menos um espaço entre "int" e "21h" e não esquecer o "h"

AnexoA – Tabela ASCII

Tabela ASCII (7bits)

Dec	Hex	Char			Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00h		^@	Null	32	20h		64	40h	@	96	60h	`
1	01h	☺	^A	SOH-Start of Header	33	21h	!	65	41h	A	97	61h	a
2	02h	☻	^B	STX- Start of Text	34	22h	"	66	42h	B	98	62h	b
3	03h	♥	^C	ETX- End of Text	35	23h	#	67	43h	C	99	63h	c
4	04h	♦	^D	EOT- End of Transmission	36	24h	$	68	44h	D	100	64h	d
5	05h	♣	^E	ENQ- Enquiry	37	25h	%	69	45h	E	101	65h	e
6	06h	♠	^F	ACK- Acknowledgment	38	26h	&	70	46h	F	102	66h	f
7	07h	●	^G	BEL- Bell	39	27h	'	71	47h	G	103	67h	g
8	08h	◘	^H	BS- Backspace	40	28h	(	72	48h	H	104	68h	h
9	09h	○	^I	HT-Horizontal Tab	41	29h	)	73	49h	I	105	69h	i
10	0Ah	◙	^J	LF-Line Feed	42	2Ah	*	74	4Ah	J	106	6Ah	j
11	0Bh	♂	^K	VT-Vertical Tab	43	2Bh	+	75	4Bh	K	107	6Bh	k
12	0Ch	♀	^L	FF-Form Feed	44	2Ch	,	76	4Ch	L	108	6Ch	l
13	0Dh	♪	^M	CR-Carriage Return	45	2Dh	-	77	4Dh	M	109	6Dh	m
14	0Eh	♫	^N	SO-Shift Out	46	2Eh	.	78	4Eh	N	110	6Eh	n
15	0Fh	☼	^O	SI- Shift In	47	2Fh	/	79	4Fh	O	111	6Fh	o
16	10h	►	^P	DLE- Data Link Escape	48	30h	0	80	50h	P	112	70h	p
17	11h	◄	^Q	DC1- (XON) Device Control1	49	31h	1	81	51h	Q	113	71h	q
18	12h	↕	^R	DC2- Device Control2	50	32h	2	82	52h	R	114	72h	r
19	13h	‼	^S	DC3- (XOFF) Device Control3	51	33h	3	83	53h	S	115	73h	s
20	14h	¶	^T	DC4- Device Control4	52	34h	4	84	54h	T	116	74h	t
21	15h	§	^U	NAK- Negative Acknowledge	53	35h	5	85	55h	U	117	75h	u
22	16h	▬	^V	SYN- Synchronous Idle	54	36h	6	86	56h	V	118	76h	v
23	17h	↨	^W	ETB- End of Trans. Block	55	37h	7	87	57h	W	119	77h	w
24	18h	↑	^X	CAN- Cancel	56	38h	8	88	58h	X	120	78h	x
25	19h	↓	^Y	EM- End of Medium	57	39h	9	89	59h	Y	121	79h	y
26	1Ah	→	^Z	SUB- Substitute	58	3Ah	:	90	5Ah	Z	122	7Ah	z
27	1Bh	←	^[	ESC- Escape	59	3Bh	;	91	5Bh	[	123	7Bh	{
28	1Ch	└	^\	FS- File Separator	60	3Ch	<	92	5Ch	\	124	7Ch	|
29	1Dh	↔	^]	GS- Group Separator	61	3Dh	=	93	5Dh	]	125	7Dh	}
30	1Eh	▲	^ ^	RS- Record Separator	62	3Eh	>	94	5Eh	^	126	7Eh	~
31	1Fh	▼	^_	US- Unit Separator	63	3Fh	?	95	5Fh	_	127	7Fh	⌂

- o sinal " ^ " antes de uma letra, significa carregar na tecla Control e simultaneamente nessa tecla
- caracteres de 0(00h) a 31(1Fh) – são caracteres especiais, que executam funções de controlo;
- carácter 32(20h) – corresponde ao código da tecla de espaço; a partir daqui e até ao carácter 126(7Eh) os caracteres têm expressão visível e podem ser impressos;
- carácter 127(7Fh) – corresponde ao código da tecla DEL (delete);
- os códigos das letras maiúsculas são inferiores aos das minúsculas diferindo por um valor igual ao código da tecla de espaço 32(20h). Ex. ASCII('a') = ASCII('A') + ASCII(' ') → 97=65+32 (61h=41h+20h);
- o código ASCII dos algarismos de 0 a 9, é dado pela soma do código do algarismo "0" que é 48(30h), mais o próprio algarismo decimal (em ASCII isso corresponde a preceder o algarismo decimal do algarismo "3"). Ex. ASCII('5') = 48+5(30h+5)=53(35h);
- o carácter 0(00h) designado por "null" é muitas vezes usado como terminador de strings (null terminated strings) e também para marcar o fim de ficheiros binários;
- ^Z – é frequentemente utilizado para marcar fim de ficheiro de texto

Tabela ASCII estendida

Dec	Hex	C	Dec	Hex	C	Dec	Hex	C	Dec	Hex	C	Dec	Hex	C
128	80h	Ç	154	9Ah	Ü	180	B4h	┤	206	CEh	╬	232	E8h	Φ
129	81h	ü	155	9Bh	¢	181	B5h	╡	207	CFh	╧	233	E9h	
130	82h	é	156	9Ch	£	182	B6h	╢	208	D0h	╨	234	EAh	Ω
131	83h	â	157	9Dh	¥	183	B7h	╖	209	D1h	╤	235	EBh	δ
132	84h	ä	158	9Eh	₧	184	B8h	╕	210	D2h	╥	236	ECh	∞
133	85h	à	159	9Fh	ƒ	185	B9h	╣	211	D3h	╙	237	EDh	φ
134	86h	å	160	A0h	á	186	BAh	║	212	D4h	╘	238	EEh	ε
135	87h	ç	161	A1h	í	187	BBh	╗	213	D5h	╒	239	EFh	∩
136	88h	ê	162	A2h	ó	188	BCh	╝	214	D6h	╓	240	F0h	≡
137	89h	ë	163	A3h	ú	189	BDh	╜	215	D7h	╫	241	F1h	±
138	8Ah	è	164	A4h	ñ	190	BEh	╛	216	D8h	╪	242	F2h	≥
139	8Bh	î	165	A5h	Ñ	191	BFh	┐	217	D9h	┘	243	F3h	≤
140	8Ch	ì	166	A6h	ª	192	C0h	└	218	DAh	┌	244	F4h	⌠
141	8Dh	Ä	167	A7h	º	193	C1h	┴	219	DBh	▉	245	F5h	⌡
142	8Eh	Å	168	A8h	¿	194	C2h	┬	220	DCh	▄	246	F6h	÷
143	8Fh	É	169	A9h	⌐	195	C3h	├	221	DDh	▌	247	F7h	≈
144	90h	æ	170	AAh	¬	196	C4h	─	222	DEh	▐	248	F8h	°
145	91h	Æ	171	ABh	½	197	C5h	┼	223	DFh	▀	249	F9h	∙
146	92h	ô	172	ACh	¼	198	C6h	╞	224	E0h	α	250	FAh	·
147	93h	ö	173	ADh	¡	199	C7h	╟	225	E1h	ß	251	FBh	√
148	94h	ò	174	AEh	«	200	C8h	╚	226	E2h	Γ	252	FCh	ⁿ
149	95h	ò	175	AFh	»	201	C9h	╔	227	E3h	π	253	FDh	²
150	96h	û	176	B0h	░	202	CAh	╩	228	E4h	Σ	254	FEh	■
151	97h	ù	177	B1h	▒	203	CBh	╦	229	E5h	σ	255	FFh	
152	98h	ÿ	178	B2h	▓	204	CCh	╠	230	E6h	µ			
153	99h	Ö	179	B3h	│	205	CDh	═	231	E7h	τ			

Nota: os caracteres da tabela estendida dependem das definições de páginas de caracteres, feitas no ficheiro config.sys (ansi.sys e country.sys)

Códigos de algumas teclas (scan codes)

Devolvidos pelo int 16h(função 10h) e int 21h(funções 06h e 07h)

Tecla	Código	Tecla	Código	Tecla	Código
F1	3Bh	Seta p/cima	48h		
F2	3CH	Seta p/baixo	50h		
F3	3Dh	Seta p/esquerda	4Bh		
F4	3Eh	Seta p/direita	4Dh		
F5	3Fh	Home	47h		
F6	40h	End	4Fh		
F7	41h	PgUp	49h		
F8	42h	PgDn	51h		
F9	43h	Insert	52h		
F10	44h	Delete	53h		
F11	7Bh				
F12	7Ah				

AnexoB - Debug

O Debug é um utilitário que permite executar um conjunto de funções de baixo-nível, tais como criar e executar pequenos programas em *assembly*, testar se um programa funciona correctamente, executando-o passo a passo, aceder directamente à memória ou ao disco, etc. Com o DEBUG é possível "espiar" o que se passa no interior da máquina.

O Debug funciona em modo DOS(Disk Operating System) pelo que é preciso abrir uma janela deste tipo para o executar. Para o fazer deve seleccionar "MS-DOS prompt" no menu "Iniciar\Programas" do Windows (isto pode variar um pouco com a versão do sistema operativo). O Debug chama-se escrevendo somente "debug" sem argumentos ou então colocando à frente o nome de um ficheiro que se quer tratar. Em qualquer caso o prompt muda para "-" indicando que o programa está à espera de comandos. Um desses comando é o "?" que é a ajuda e mostra os comandos disponíveis. Outro comando é o "Q" que permite voltar ao DOS; uma vez aí retorna-se ao Windows escrevendo "exit".

Verificar os códigos ASCII gravados num ficheiro de texto

Usando um editor de texto qualquer (ex: NotePad do Windows), crie na directoria "c:\tmp" (ou noutra qualquer) um ficheiro de texto apenas com a palavra "hello", guardando-o com o nome "teste.txt". Entre em modo DOS e chame o Debug escrevendo "Debug teste.txt". Dê o comando "d" (dump) e veja o resultado. Na coluna da esquerda aparecem os endereços de memória aonde o ficheiro foi carregado, no formato nnnn:nnnn em sequência crescente; na coluna central aparecem os códigos ASCII correspondentes ao texto que está á direita, no qual deverá estar a palavra "hello" (além da palavra "hello aparece mais texto que não pertence ao ficheiro que criou, mas sim a outros).

Verificar a utilização do modo "little endian" de armazenamento em memória

Este teste pode ser efectuado de dois modos. O primeiro é criar um ficheiro aonde tenha sido gravado um valor numérico, p.ex., o inteiro 741 (02E5h). Isto pode ser feito através de um programa escrito numa linguagem de alto-nível como C ou Pascal ou então em *assembly*. Como isso pode não ser fácil neste momento, pode usar-se de novo o método anterior mas usando o comando "u" (unassemble) em vez do anterior. O Debug mostra novamente na coluna da esquerda os endereços de memória aonde foi armazenado o código, no formato nnnn:nnnn em sequência crescente; na segunda coluna aparece o código-máquina da instrução que consta da terceira e quarta colunas. Relativamente ao ficheiro de teste, verifique que os primeiros seis códigos correspondem mais uma vez aos da palavra "hello". Em seguida continue a dar comandos "u" até surgirem instruções do tipo "MOV ... , valor numérico", verificando que esse valor numérico aparece invertido na coluna correspondente ao código-máquina que foi armazenado.

AnexoC - Métrica binária

Pessoas	Computadores
Fácil reconhecimento de símbolos, estados ou níveis	Apenas dois estados básicos: ligado(0) e desligado(1)
Base decimal (B=10)	Base binária (B=2)
10 símbolos(alagarismos): 0, .. ,9	2 símbolos(algarismos): 0 , 1 [*bit*]

De acordo com a expressão de significância posicional que rege os sistemas de numeração, tem-se que para um código binário constituído por <u>*n*</u> bits, o seu valor em decimal (base 10) é:

$$M_{10} = \sum_{i=0}^{n} A_i * B^i = A_n * B^n + A_{n-1} * B^{n-1} + ... + A_1 * B^1 + A_0 * B^0$$

em que B é a base de numeração (B=2 em binário), A_i são os algarismos da base (0 ou 1 em binário), B^i o peso ou significância posicional do algarismo $A_{i,}$ sendo i o índice posicional do algarismo.

Para o caso de n=1, 2, 3 e 4, obtém-se:

decimal	**binário** (pesos) $2^3\ 2^2\ 2^1\ 2^0$ 8 4 2 1	**hexadecimal**	**n**
0	0 0 0 0	0	n=1 (2 casos) / n=2 (4 casos) / n=3 (8 casos) / n=4 (16 casos)
1	0 0 0 1	1	
2	0 0 1 0	2	
3	0 0 1 1	3	
4	0 1 0 0	4	
5	0 1 0 1	5	
6	0 1 1 0	6	
7	0 1 1 1	7	
8	1 0 0 0	8	
9	1 0 0 1	9	
10	1 0 1 0	A	
11	1 0 1 1	B	
12	1 1 0 0	C	
13	1 1 0 1	D	
14	1 1 1 0	E	
15	1 1 1 1	F	

Note-se que para um certo número n de bits o maior valor decimal representado é 2^n-1

Alguns exemplos com interesse em tecnologia dos computadores são:

n	**gama=2^n**	**designação**	**nº de bytes**
1	2 : (0 , 1)	bit	-
4	16 : (0 .. 15)	nibble	-
8	256 : (0 .. 255)	byte (octeto)	1
10	1024 : (0 .. 1023)	Kbit	-
16	65536 : (0 .. 65535)	Palavra 16 bits (word)	2
32	2^{32} : (0 .. 2^{32}-1)	Palavra 32 bits	4
64	2^{64} : (0 .. 2^{64}-1)	Palavra 64 bits	8

<u>NOTA</u>: a definição de "word" varia consoante a máquina usada. Em máquinas com registos de 16 bit, word=16 bit; máquinas com registos de 32 bit, word=32 bit

Dentre os exemplos anteriores, o *byte* (binary term), também designado por octeto, assume grande importância, uma vez que historicamente tem sido utilizado como o grupo mínimo de bits usado para representar a informação. Efectivamente, o tratamento de dados na forma digital não é habitualmente feito usando-se os bits 0 e 1 individualmente (embora haja casos em que isso acontece), mas sim em grupos, de que o byte (8 bits) é o mais universalmente usado. Por sua vez os grupos *nibble* (metade de um *byte*) e *word* (dois *byte*) são também muito utilizados.

O caso *n*=10 (Kbit) assume especial importância. O prefixo K significa $1000=10^3$ em decimal, em que 10 é a base da numeração decimal e 3 é o expoente a que esta deve ser elevada para obter 1000. De acordo com este princípio, para em binário obtermos o valor de K, deveríamos elevar a base 2 a um expoente tal que se obtivesse 1000, ou seja $1000=2^n$, só que não existe nenhum expoente inteiro que verifique aquela relação; o valor mais próximo é 10, obtendo-se $2^{10}=1024$, pelo que em binário K=1024.

A partir da unidade byte definem-se os seus múltiplos, entre os quais:

binário	**gama**	**unidade**	**decimal aproximado**
2^{10}	1024 byte	K (Kilo) - Kbyte	10^3
2^{20}	1024 * K = 1 048 576 byte	M (Mega) - Mbyte	10^6
2^{30}	1024 * M = 1 073 741 824 byte	G (Giga) - Gbyte	10^9
2^{40}	1024 * G = 1 099 511 627 776 byte	T (Tera) - Tbyte	10^{12}

Tabela das potências de 2

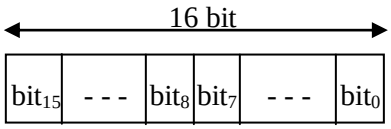
n	**2^n(decimal)**	**2^n(hexadecimal)**	**unidade**
0	1	1	
1	2	2	
2	4	4	
3	8	8	
4	16	10h	
5	32	20h	
6	64	40h	
7	128	80h	
8	256	100h	
9	512	200h	
10	1024	400h	1K
11	2048	800h	2K
12	4096	1000h	4K
13	8192	2000h	8K
14	16384	4000h	16K
15	32768	8000h	32K
16	65536	10000h	64K

Para cada valor de ***n*** a gama de representação é dada pelo intervalo [0 .. 2^n-1]

AnexoD - Registos do 8086

Agrupam-se em: 1)dados 2)endereços 3)segmento 4)controlo

1)DADOS (uso geral e funções do processador) X=16bit H=High(8bit) L=Low(8bit)



			Funções típicas
AX	AH	AL	Acumulador: operações lógicas/aritméticas, I/O, funções do processador , etc
BX	BH	BL	Base: modos de endereçamento
CX	CH	CL	Contagem: contagens, ciclos (loops)
DX	DH	DL	Dados: multiplicação/divisão, I/O, extensão do AX

2)ENDEREÇOS

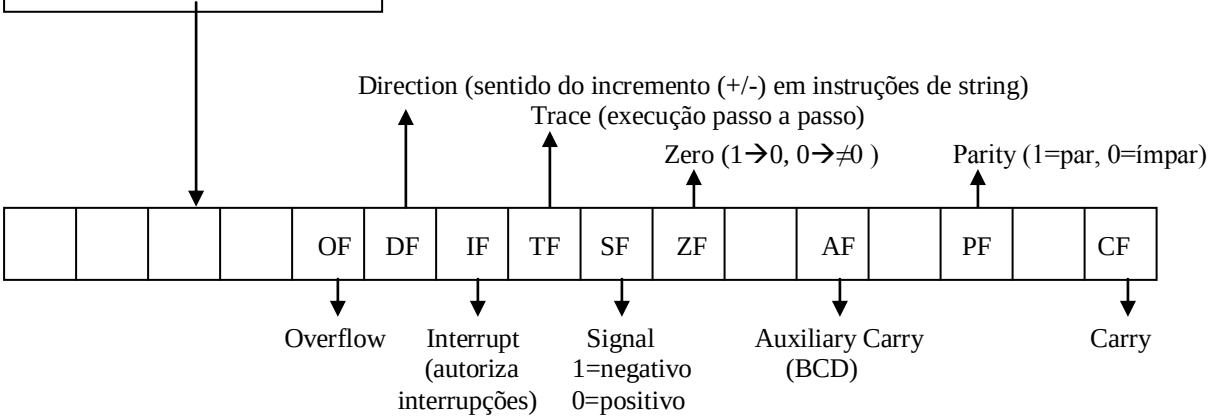
SP	Stack Pointer: controle da pilha (stack)
BP	Base Pointer: registo base de endereçamento
SI	Source Index: indexador de origem (strings)
DI	Destination Index: indexador de destino (strings)

3)SEGMENTO

CS	Code Segment: acesso ao código
DS	Data Segment: acesso aos dados
SS	Stack Segment: acesso à pilha
ES	Extra Segment: extensão (strings)

4)CONTROLO

IP	Instruction Pointer: indica a localização das instruções na memória
FLAGS	Indicadores do estado do processador



As flags indicam o estado do processador, sendo afectadas por certas operações como as aritméticas e lógicas. TF, IF e DF são flags de comando (são instruções do utilizador ao processador); as restantes são flags de *status* (destinam-se a serem lidas para testar a ocorrência do acontecimento respectivo).

AnexoE – Interrupções do DOS e da BIOS

DOS – Disk Operating System			
int 21h – Diversas funções do DOS			
Função	Descrição	Registos	
06H	Obtém um carácter do teclado e mostra-o no ecrã. Não espera que se carregue numa tecla. No caso de teclas especiais, (F1,F2,...), o registrador AL conterá o valor 0, sendo necessário chamar a função novamente para obter o código da tecla. O carácter 255 não pode ser usado pois significa "receber"	Entrada	AH=06H DL=0-254 Envia o código da tecla DL=FFH(255) Recebe um carácter
		Saída	Se DL=0-254, não tem retorno algum Se DL=255: ZF=1 → não houve leitura ZF=0 → AL=carácter lido
07H	Obtém um carácter do teclado sem o mostrar no ecrã. Espera que se carregue numa tecla. No caso de teclas especiais, (F1,F2,...), o registrador AL conterá o valor 0, sendo necessário chamar a função novamente para obter o código da tecla.	Entrada	AH=07H
		Saída	AL=carácter lido
3CH	Cria um ficheiro (handle)	Entrada	AH=3CH CX=atributo (0= ficheiro normal) DS=segmento (desnecessário em FASM) DX= path para o ficheiro (ASCIIZ)
		Saída	CF=0 → OK → AX=handle do ficheiro CF=1 → Erro → AX= 3 – path não encontrado 4 – sem handles disponíveis 5 – acesso não permitido
3DH	Abre um ficheiro (handle)	Entrada	AH=3DH AL=método de acesso: 00H – só leitura 01H – só escrita 02H – leitura/escrita DS=segmento (desnecessário em FASM) DX= path para o ficheiro (ASCIIZ)
		Saída	CF=0 → OK → AX=handle do ficheiro CF=1 → Erro → AX= 1 – função inválida 2 – ficheiro não encontrado 3 – path não encontrado 4 – sem handles disponíveis 5 – acesso não permitido 12 – código de acesso inválido
3EH	Fecha ficheiro (handle) NOTA: não indicar valores de handle 0..4 pois referem-se a ficheiros de sistema que uma vez fechados não podem ser acedidos.	Entrada	AH=3EH BX=handle do ficheiro
		Saída	CF=0 → OK → AX=handle do ficheiro CF=1 → Erro → AX= código de erro

3FH	Lê de um dispositivo ou ficheiro(handle) (handle = ponteiro para o dispositivo ou ficheiro)	Entrada	AH=3FH BX=handle do ficheiro (0=teclado) CX=número de bytes a ler DX=endereço da variável de leitura
		Saída	CF=0 → AX=número de bytes lidos CF=1 → Erro → AX= código de erro
40H	Escreve em um dispositivo ou ficheiro(handle) (handle = ponteiro para o dispositivo ou ficheiro)	Entrada	AH=40H BX=handle do ficheiro (1=ecrã) CX=número de bytes a escrever DX=endereço da variável de escrita
		Saída	Se CF=0, AX=número de bytes escritos CF=1, Erro → AX= código de erro
4CH	Termina a execução de um programa	Entrada	AH=4CH
		Saída	AL= código de retorno (para chamadas dentro de ficheiros batch, pode ser testado por ERRORLEVEL)

BIOS – Basic Input/Output System			
int 12h - Memória			
Função	Descrição	Registos	
	Obtém a quantidade total de memória do sistema. PC , XT → Mem = 640 Kbyte AT → não inclui memória de vídeo nem memória estendida	Entrada	
		Saída	AX= quantidade de memória em blocos contínuos de 1 Kbyte
int 15h – Serviços do sistema			
Função	Descrição	Registos	
88H	Obtém a quantidade de memória estendida do sistema (acima de 1Mb=1024Kb)	Entrada	AH=88H
		Saída	AX= quantidade de memória em blocos contínuos de 1 Kbyte acima de 1 Mb
int 16h - Teclado			
Função	Descrição	Registos	
00H	Obtém um carácter do teclado, sem o mostrar no ecrã. Devolve o "scan code" da tecla e o seu código ASCII se tiver algum. Funciona também para as teclas especiais (F1,F2,...).	Entrada	AH=00H
		Saída	AH=scan code (código da tecla no teclado) AL=código ASCII da tecla ou E0H(?) se a tecla é especial (F1,F2,...)
01H	Verifica se no buffer do teclado existe algum carácter para ser lido. Não remove o carácter; para o fazer usar a função 10H. Funciona também para as teclas especiais (F1,F2,...).	Entrada	AH=01H
		Saída	CF=0 → existe carácter AH=scan code (código da tecla no teclado) AL=código ASCII da tecla ou E0H(?) se a tecla é especial (F1,F2,...)

02H	Obtém o status do teclado.	Entrada	A=02H
		Saída	AX → bits contêm estado das teclas: Bit Significado se for 1 00 Shift direito pressionado 01 Shift esquerdo pressionado 02 Control pressionado 03 Alt pressionado 04 Scroll Lock ligado 05 Num Lock ligado 06 Caps Lock ligado 07 Insert ligado 08 Control esquerdo pressionado 09 Alt esquerdo pressionado 10 Control direito pressionado 11 Alt direito pressionado 12 Scroll Lock pressionado 13 Num Lock pressionado 14 Caps Lock pressionado 15 Sys Req pressionado

int 10h – Vídeo			
Função	Descrição	Registos	
00H	Define o modo de vídeo standard. Provoca a limpeza do ecrã. Em modo gráfico, o cursor não é mostrado.	Entrada	AH=00H AL=modo vídeo (Ver tabela)
		Saída	Destrói : AX, SP, BP, SI, DI
4FH	Define o modo de vídeo SVGA (VESA). Esta função pode usar-se em vez de 00H.	Entrada	AH=4FH AL=02H BX=modo vídeo (Ver tabela)
		Saída	AH= 00H→ OK 01H→Erro AL=4FH em caso de sucesso
02H	Define a posição do cursor no ecrã. A origem (0,0) é o canto superior esquerdo do ecrã.	Entrada	AH=02H BH=nº da página de vídeo(0=modos gráficos) DH=nº da linha DL=nº da coluna
		Saída	Destrói: AX, SP, BP, SI, DI
0AH	(Texto/Gráfico) Escreve um carácter no ecrã, na posição do cursor. Não altera a posição do cursor.	Entrada	AH=0AH AL=código ASCII do carácter a escrever BL=cor do fundo em modo gráfico BH=nº da página de vídeo(0=modos gráficos) CX=nº de repetições do carácter
		Saída	Destrói: AX, SP, BP, SI, DI

<table>
<tr><td rowspan="2">0BH</td><td rowspan="2">Estabelece Palette de cores.
(Não funciona em todos os modos de vídeo)</td><td>Entrada</td><td>AH=0BH

BH=00H → define cores da borda (border) e do fundo (background)
BL=código de cor

BH=01H → selecciona pallete de cores
BL=código da pallete

Pallete Pixel Color
0 0 mesma do fundo
1 verde
2 vermelho
3 amarelo
1 0 mesma do fundo
1 azul
2 magenta
3 branco</td></tr>
<tr><td>Saída</td><td>Destrói: AX, SP, BP, SI, DI</td></tr>
<tr><td rowspan="2">0CH</td><td rowspan="2">Desenha um pixel no ecrã.

<u>Legal values</u>
Mode CX DX Pixel Color BH
10H 0-639 0-349 0-15 0-1
13H 0-319 0-199 0-255</td><td>Entrada</td><td>AH=0CH
AL=cor (atributo) do pixel
BH=nº da página gráfica (0)
CX=coluna
DX=linha</td></tr>
<tr><td>Saída</td><td>Destrói: AX, SP, BP, SI, DI</td></tr>
<tr><td rowspan="2">10H</td><td rowspan="2">Controla operações nos registos de cor.</td><td>Entrada</td><td>AH=10H

AL=10H→ define registo de cor
BX=cor a definir
DH=valor de Red (R)
CH=valor de Green (G)
CL=valor de Blue (B)

AL=15H→ obtém registo de cor
BX=cor a definir

<u>Saída:</u>
DH=valor actual de Red (R)
CH=valor actual de Green (G)
CL=valor actual de Blue (B)</td></tr>
<tr><td>Saída</td><td>Destrói: AX, SP, BP, SI, DI</td></tr>
</table>

Handles pré-definidos do DOS		
Handle	**Designação**	**Notas**
0	Standard Input Device - can be redirected (STDIN)	Teclado
1	Standard Output Device - can be redirected (STDOUT)	Ecrã
2	Standard Error Device - can be redirected (STDERR)	Ecrã
3	Standard Auxiliary Device (STDAUX)	(Porta série)
4	Standard Printer Device (STDPRN)	Impressora

Tabela de cores (código - cor)			
0 - BLACK	4 - RED	8 - DARKGRAY	12 - LIGHTRED
1 - BLUE	5 - MAGENTA	9 - LIGHTBLUE	13 - LIGHTMAGENTA
2 - GREEN	6 - BROWN	10 - LIGHTGREEN	14 - YELLOW
3 - CYAN	7 - LIGHTGRAY	11 - LIGHTCYAN	15 - WHITE

Códigos de erro (devolvidos em AX)	
Código	**Descrição**
01H	Invalid function number
02H	File not found
03H	Path not found
04H	Too many open files (no handles left)
05H	Access denied
06H	Invalid handle (ex. file not open)
07H	Memory control blocks destroyed
08H	Insufficient memory
09H	Invalid memory block address
0AH	Invalid environment
0BH	Invalid format
0CH	Invalid access mode (open mode is invalid)
0DH	Invalid data
0EH	Reserved
0FH	Invalid drive specified
10H	Attempt to remove current directory
11H	Not same device
12H	No more files

Table of Video Modes					
Mode (AL)	**Type**	**Resolution**	**Adptader(s)**	**Colors**	**Address Buffer**
00H	Text	40 x 25	CGA,EGA,MCGA,VGA	B/W 16 gray	B8000
…	…	…	…	…	…
03H	Text	80 x 25	CGA,EGA,MCGA,VGA	16 fore/8 back	B8000
…	…	…	…	…	…
10H	Graphics	640 x 350	EGA,VGA	16 color	A0000
…	…	…	…	…	…
13H	Graphics	320x200	MCGA,VGA	256 color	
Modos SVGA (VESA)					
…	…	…	…	…	…
10FH	High Res	320 x200		16,777,216	A000
112H	High Res	640 x480		16,777,216	A000
115H	High Res	320 x200		16,777,216	A000
118H	High Res	1024 x768		16,777,216	A000
11BH	High Res	1280 x1024		16,777,216	A000

NOTAS: 1) os modos marcados a cinza serão os utilizados nas aulas práticas (modo 03h = default)

2) modos acima de 10CH não são suportados por todas as cartas VGA

DOSBox – utilizado para executar exemplos do FASM (corre em qualquer sistema operativo)

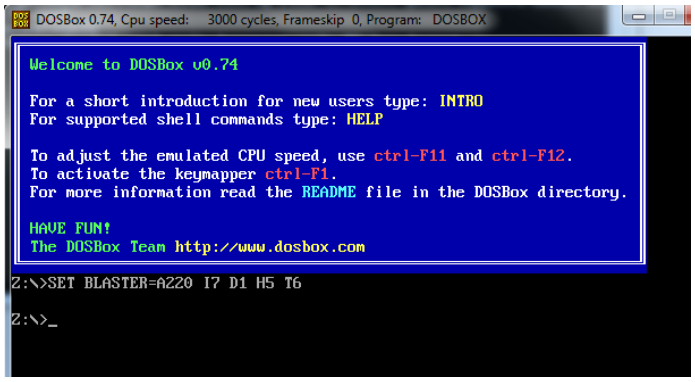
https://pt.wikipedia.org/wiki/DosBox

O DOSBox permite emular (ou simulador) um PC baseado no Intel x86, incluindo som, gráficos e mouse entre outros.

Download: https://www.dosbox.com/
https://sourceforge.net/projects/dosbox/
(versão portátil: https://sourceforge.net/projects/portableapps/files/latest/download)

1) Descarregar e instalar o DOSBox (versão atual, 0.74)

2) Executar o DOSBox → cria uma janela DOS como abaixo e posiciona-se no drive Z



3) Montar um drive aonde serão executados os programas → Z:\>mount <drive> <path>

<drive> = C , D , ... (não usar A , B , Z) <path> = caminho para uma diretoria já existente

exemplo:

Z:\>mount C C:\temp (nas máquinas da 6.15, mount Z:\>C C:\users\<grupo>…)
Z:\> C:
C:\>

- listar ficheiros da diretoria → C:\> dir
- executar um ficheiro (“.exe” , “.com” , “.bat”) → C:\> <file> ex: C:\> teste.com
- limpar o écran → C:\>cls (clear screen)
- sair do DOSBox → C:\> exit (ou fechar a janela)

FASMW ↔ DOSBox

1) criar um programa em Assembly/FASM: ex. “teste.asm”

2) em seguida Run>Compile → cria o “teste.com” (sem executar)

3) copiar “teste.com” para a directoria local aonde se fez o mount no DOSBox:
ex: “teste.com” → C:\temp

4) executar o programa → C:\> teste ou C:\> teste.com