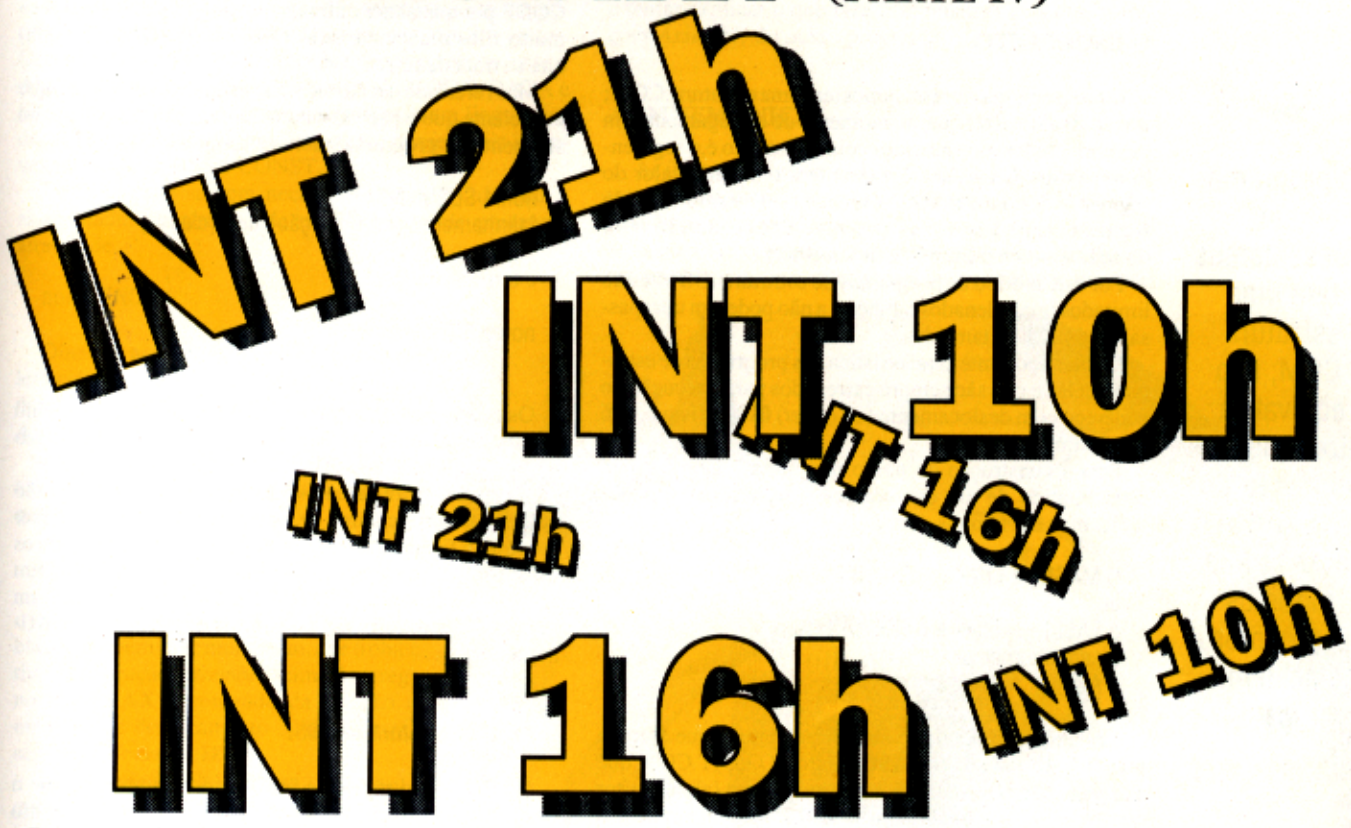
PROGRAMAR EM ASSEMBLY (PARTE IV)



Durante os três primeiros artigos sobre linguagem *Assembly* foi nossa principal preocupação tentar «preparar o terreno» para que o leitor menos conhecedor dos «meandros» internos do PC, poder vir a ter sucesso no estudo desta importante linguagem de programação.

Mas agora chegou (finalmente) a altura de nos debruçarmos sobre os aspectos da programação propriamente dita.

O leitor ainda deverá estar recordado que prometemos no artigo anterior que iríamos desenvolver o tema dos *interrupts* através de um pequeno programa que deles fizesse uso. Não se trata de nada de excepcional, pois efectivamente mais difícil seria fazer um programa em *Assembly* que não utilizasse os ditos *interrupts*, já que estes permitem, entre outras coisas, aceder a um vasto «manancial» de rotinas existentes quer no DOS, quer no BIOS.

Os *interrupts* mais utilizados para essa finalidade são os *software interrupts* (referidos no artigo anterior) e as respectivas rotinas estão amplamente documentadas em muitos livros sobre a programação em *Assembly* e também em outras linguagens. De entre todos esses *interrupts*, o mais pródigo em rotinas úteis é o **INT 21h** e recebe o prestigioso nome de *DOS Functions* (ou aqui para nós, Funções DOS).

Seleccionamos uma das Funções DOS carregando o registo **AH** com o valor correspondente a essa Função e, como é frequentemente necessário passar também parâmetros à função (aliás o mesmo aconteceria nas funções da matemática), os respectivos valores são carregados nos registos adequados (e de acordo com o previsto pela sintaxe dessa função), depois disso tudo feito emite-se um **INT 21h** e aguarda-se (uns microsegundos) pela resposta.

Duas Funções DOS foram escolhidas para fazer parte do nosso exemplo, a função **2Ch** (que foi baptizada com o nome de GET TIME) e a função **09h** (com o nome de DISPLAY STRING).

Um outro *interrupt* muito útil, o **INT 10h**, também consta do nosso pequeno programa. Este *interrupt* é designado por *ROM BIOS Video Services* ou simplesmente Video_IO (sendo IO abreviatura de *Input Output*).

No nosso programa utilizamos os serviços **02h** (*Set Cursor Position*) e **03h** (*Read Cursor Position*).

E como não há dois sem três, incluímos ainda um outro *interrupt*,: o **INT 16h**, o qual recebe o nome de *ROM BIOS Keyboard Services* ou apenas Keyboard_IO.

O serviço **01h** do **INT 16h** designa-se por *Report Whether Character Ready* (isto é, propõe-se informar se está algum carácter no *buffer* do teclado à espera de ser lido).

O nosso exemplo ilustra bem a sintaxe das funções que se baseiam nos *interrupts* que acabamos de apresentar, e consideramos ser desnecessário detalhá-las mais, não só por limitações de espaço, e porque é muito fácil compreender o que fazem, mas muito particularmente porque existem literalmente milhares de outras rotinas diferentes que podem ser chamadas através de *interrupts*.

O mais completo «tratado» sobre quais são as rotinas acedíveis por *interrupts*, o que fazem e como obter os seus serviços é um ficheiro electrónico de nome INTERRUPT.LST de um senhor chamado Ralph Brown e cuja última revisão que temos conhecimento ocupa nada menos que 1,7 MB em disco após a descompressão (podendo por conseguinte ultrapassar as 700 páginas se for impresso ...).

É distribuído em regime *shareware* pelas BBS, e recomendamos que o tente obter «a todo o custo» (dica: experimente ver se faz parte da «Montra Spooler»...).

> “... De entre todos esses interrupts, o mais pródigo em rotinas úteis é o INT 21h e recebe o prestigioso nome de DOS Functions...”

Vamos agora tratar de dois temas muito importantes para se poder compreender o funcionamento do programa que aqui reproduzimos. O primeiro é construir um «esqueleto» (este não irá ter ossos) e o segundo é, muito naturalmente, explicar ao leitor o que fazem mnemónicas *Assembly* do tipo das que estão presentes no programa.

Vamos começar pelo esqueleto.

O ESQUELETO

> O programa que apresentamos tem uma estrutura .COM; isto equivale a dizer que, no momento do carregamento em memória, todos os segmentos coincidem...

O programa que apresentamos tem uma estrutura .COM; isto equivale a dizer que, no momento do carregamento em memória, todos os segmentos coincidem, isto é,o segmento de código **CS** (*code segment*) tem o mesmo valor do segmento de dados **DS** (*data segment*), o mesmo valor do segmento extra **ES** (*extra segment*) e ainda o mesmo valor do segmento de pilha **SS** (*stack segmet*).

Por este motivo os programas de estrutura .COM estão limitados (e condenados também) a não poderem ultrapassar os 64 KB de tamanho.

Outra importante característica dos programas de estrutura .COM é que são sempre carregados para execução no endereço com deslocamento (ou *offset*) **0100h** (isto é, 256 em decimal).

Para a construção do nosso programa o leitor deverá estar munido de um dos conjuntos de ferramentas que indicámos em tempos, isto é:

```
O MASM, o LINK e o EXE2BIN
ou
o TASM, o TLINK e o EXE2BIN
ou em alternativa
o A86
```

Deverá dispor ainda de um editor de texto que escreva texto não formatado (ou ASCII puro). O EDIT.COM que acompanha o MS-DOS 5.0 serve muito bem (algum leitor que se sinta à vontade com o EDLIN.COM também o poderá utilizar, embora hesitemos em o recomendar).

O exemplo foi elaborado para ser assemblado sem qualquer modificação quer pelo MASM, quer pelo TASM, quer ainda pelo A86. Neste exemplo, quem dispuser do A86 fica com a vantagem acrescida de prescindir da fase de ligação, quer pelo LINK quer pelo TLINK, contudo fica o leitor alertado que nem sempre tal é possível (seria efectivamente bom de mais).

O EXE2BIN também é normalmente dispensável com ficheiros de estrutura .COM para quem utilize a dupla TASM, e TLINK com o *switch* **/t**.

Com estas ideias bem claras vamos então apresentar um esqueleto que possa servir para assemblar qualquer programa com estrutura .COM.

Ei-lo:

```
codigo SEGMENT
ASSUME cs:codigo, ds:codigo, es:codigo,
ss:codigo
ORG 100h

codigo ENDS
END inicio
```

Este é o esqueleto clássico, ou obtido com **directivas convencionais**. O MASM e o TASM actuais aceitam alegremente um outro tipo de directivas designadas por **directivas simplificadas** e que são baseadas nos modelos de memória das linguagens de alto nível.

Para os programas de estrutura .COM seria qualquer coisa como:

```
.MODEL TINY
.CODE
ORG 100h

END inicio
```

O A86 não compreende este último tipo de directivas, contudo e pelo menos na versão que dispomos, assembla sem erro (aliás podíamos ter substituído MODEL TINY e CODE por quaisquer outras palavras) pois este «assume» muito filosóficamente pelo **ORG 100h** (ORiGin 100h) que se trata de um .COM.

Mas referindo-nos de novo ao esqueleto clássico, fique consciente que o pseudo-operador para a definição de um segmento de programa inicia-se sempre com:

```
nome SEGMENT
[alinhamento][combinação][uso][classe]
```

e termina com:

```
nome ENDS
```

Os programas de estrutura .COM só têm um segmento de programa, mas os ficheiros de estrutura .EXE poderão ter tantos quantos necessitemos ou desejemos.

As opções entre parêntesis rectos normalmente não são utilizadas com ficheiros de estrutura .COM; de qualquer modo aqui ficam as principais pois são importantes para os ficheiros de estrutura .EXE (para o assemblador, um ficheiro de estrutura .COM é um caso particular de um ficheiro de estrutura .EXE), conforme veremos no próximo artigo.

Opções de *alinhamento*:

PARA – É o *default* e informa o assemblador que o segmento de programa se deve iniciar em um parágrafo (múltiplo de 16 bytes).

BYTE – O segmento de programa pode-se iniciar em qualquer endereço.

WORD – O segmento de programa inicia em endereço par.

PAGE – O segmento de programa inicia em endereço múltiplo de 256.

– Opções de *combinação*:

AT "expressão" – Informa que esse segmento está localizado no parágrafo definido pela "expressão". O assemblador não gerará código para esse segmento; esta opção apenas visa facilitar a referenciação de dados existentes em endereços fixos da memória.

PRIVATE – É o *default*, e faz com que o assemblador não combine esse segmento com outro mesmo que ambos tenham o mesmo nome.

PUBLIC – Todos os segmentos com o mesmo nome são reunidos de modo a ficarem a formar um único segmento. Esta opção é normalmente utilizada quando se assemblam separadamente vários módulos de um programa que deverão ser combinados pelo *linker*.

STACK – Quando esta opção é referenciada, logo após o carregamento em memória do programa o registo de segmento **SS** apontará para esse segmento de programa e o registo **SP** apontará para o fim dele.

– Opções de *uso*:

São utilizadas para programas que executam apenas em processadores 80386SX ou superior, pelo que fique o leitor só a saber que existem e que um dia pode vir a precisar de as usar.

– Opções de *classe:*

Controlam a ordem e o lugar dos segmentos em memória. Segmentos com a mesma classe são colocados juntos. O nome da classe é rodeado por apóstrofes ou aspas. Alguns nomes usuais para as classes são CODE, DATA e STACK, embora quem tenha experiência em linguagens de alto nível talvez já tenha lido outros nomes mais «arrevezados» como BSS e CONST.

Finalmente, é sempre necessário também informar o assemblador do término de um segmento de programa através de:

```
nome ENDS
```

Olhando novamente para o nosso esqueleto clássico, vemos um dos pseudo-operadores que mais «má fama» têm, o célebre **ASSUME**.

Imagine o leitor que «estava na pele» do assemblador, e que lhe aparecia a instrução *assembly* seguinte:

```
mov ax, [ 200h ]
```

o leitor sabe, (pois já o referimos) e o assemblador também sabe muito bem, que o objectivo da instrução *Assembly* acima é carregar no registo **AX** o conteúdo da posição de memória de deslocamento **200h** dentro do segmento de dados **DS**.

Efectivamente, ele sabe muito bem isso, o que ele pode ter dúvidas, e ao ponto de o fazer hesitar na produção do código máquina adequado, é para qual dos segmentos do programa o registo de segmento **DS** está a apontar naquele momento. E aqui surge uma das razões principais da necessidade do ASSUME: ajudar o assemblador na produção de código máquina.

E agora dirão alguns leitores, que sabem muito bem ou têm ouvido dizer, que o A86 dispensa os ASSUME e, por conseguinte, este assemblador parece mais evoluído que o MASM e o TASM.

Efectivamente o A86 poderá com alguma segurança dispensar os ASSUME em programas de estrutura .COM (área em que o A86 é especialista).

E as versões mais recentes do MASM e do TASM também dispensam os ASSUME, normalmente sem perigo, em ficheiro de estrutura .COM...

Em resumo, é de boa política manter sempre os ASSUME, mas quem lhes for adverso poderá muitas vezes (existem excepções) eliminá-los em ficheiros de estrutura .COM.

Finalmente, e olhando mais uma vez para o esqueleto clássico, vemos que tudo termina com um aparentemente contraditório:

```
END inicio
```

O operando "inicio" (poderia ter qualquer outro nome) da directiva **END** é o nome de um *label* (ou *procedure*) definido algures dentro do programa e que marcou o ponto de entrada do mesmo. Nos ficheiros de estrutura .COM a primeira instrução *Assembly* deve estar antecedida desse *label* ou então pertencer a um *procedure* com esse nome (conferir pelo nosso programa). E já dissemos ao leitor o essencial sobre esqueletos para produzir tantos ficheiros .COM quantos deseje.

Vamos agora cumprir a segunda parte da nossa missão e tentar explicar o que fazem as mnemónicas das instruções *Assembly* do nosso programa.

AS MNEMÓNICAS

As instruções que constituem o programa são compostas por mnemónicas das mais frequentemente utilizadas pela linguagem *Assembly*, e, tão vulgares elas são, que seria possível construir muitos outros programas diferentes restringindo-nos apenas a este pequeno e limitado

grupo. Todas as instruções *Assembly* são compostas pela mnemónica propriamente dita seguida de zero, um ou dois operandos.

Uma instrução com zero operandos é, por exemplo, a instrução **RET.**

Quando o processador encontra esta instrução, retira do topo da pilha um *word* e coloca-o no registo **IP**. O resultado é a execução ser transferida para a instrução apontada pelo valor do registo **IP** dentro do segmento de código **CS**.

O oposto da instrução **RET** é a instrução **CALL *Subrotina.*** Esta instrução guarda na pilha o valor de deslocamento da instrução que se lhe segue dentro do segmento de código **CS** e deposita no registo **IP** o valor do deslocamento do ponto de destino.

Nota: Se o *procedure* alvo fosse do tipo *far*, o valor **CS** também iria para a pilha antes do processador depositar no registo **CS** o valor do novo segmento.

Como se vê, as instruções **RET** e **CALL *Subrotina*** proporcionam dois meios de alterar o fluxo de execução de um programa, mas existem outras alternativas:

– O salto incondicional **JMP *label***

Aqui a transferência do fluxo de execução é feita carregando sem mais delongas no registo **IP** o valor do deslocamento de *label* dentro do segmento **CS** (em *procedures near*).

– O salto condicional **J(condição) *label***

No nosso exemplo vemos a variante **JZ** (salta na condição de o bit **Zero Flag** ser **1**).

Muitas outras possiblidades existem tais como **JC** *(jump if carry flag)*, **JA** *(jump if above)*, **JO** *(jump if overflow flag)* , todas elas relacionadas com o estado das várias *flags*, excepto uma: **JCXZ (jump if CX = 0)**.

A mnemónica que mais se utiliza, não só neste mas em quase todos os programas, é a **MOV**, e apresenta-se em várias variantes consoante o modo de transferência\endereçamento que está a ser efectuado e este pode ser por exemplo:

–Entre registos
Um registo recebe o valor existente em um outro.
Como é o caso de: **MOV AL,CH**

–Dado Imediato para o Registo
É transferido para um registo um dado, presente no segundo operando da instrução. E aqui é o caso de: **MOV AH, 2Ch**

– Directo do Registo para a Memória
Um endereço de memória (dado normalmente pelo nome de uma variável) recebe o valor existente num registo. Como em: **MOV [MESMO_SEG], DH**

Nota: Alguns assembladores dispensam os parêntesis rectos à volta do nome das variáveis.

– Do acumulador para uma posição de memória indexada pelos registo SI ou DI.
É o que se passa com o nosso: **MOV [SI], AL**

Repare agora na instrução do programa:

```
MOV SI, OFFSET [HORAS + 1]
```

OFFSET é um operador interno do assemblador que nos permite fazer com que ele calcule para nós (neste caso) o valor do deslocamento correspondente à posição de memória do segundo byte da variável HORAS (também serviria para calcular o deslocamento de *labels* se fosse caso disso).

> “Todas as instruções Assembly são compostas pela mnemónica propriamente dita seguida de zero, um ou dois operandos...”

Outras duas mnemónicas muito úteis são **OR** e **XOR**.

OR executa (como seria de esperar) a operação lógica de nome “OU” bit a bit entre dois operandos e devolve o resultado no primeiro deles afectando alguns *flags*, designadamente o *Zero Flag*. Ex: OR AL, AL

Aqui,quando **AL** for nulo o resultado também é nulo, e o **Zero Flag** vem igual a **1**, o que nos permite (vd. nosso exemplo) que a instrução de salto condicional que se lhe segue nos conduza para o label **SAIDA**.

XOR executa um “OU EXCLUSIVO” bit a bit entre dois operandos. Uma das utilidades do XOR é pôr um registo a zero como em:

```
XOR AL, AL
```

A mnemónica **DEC** subtrai uma unidade ao conteúdo do operando como em: DEC SI

O CPU é capaz de efectuar divisões quer com números só positivos (mnemónica **DIV**), quer com números positivos e negativos (mnemónica **IDIV**).

No nosso programa ficámo-nos pela primeira dessas possibilidades:

DIV BL divide o conteúdo do registo **AX** pelo do registo **BL**, o quociente virá em **AL** e o resto em **AH**.

Finalmente, a mnemónica **CMP** compara dois operandos e o resultado é passado para as *flags* adequadas.

Existem quase tantas alternativas para a mnemónica CMP como para a MOV, contudo no nosso exemplo ficámo-nos pela:

```
CMP DH,[MESMO_SEG]
```

que compara o conteúdo do registo **DH** com o conteúdo da posição de memória dada pela variável MESMO_SEG dentro do segmento **DS**.

Para além das mnemónicas o leitor irá reparar que antecedemos todos os dados com **DB** (Define Byte) ou **DW** (Define Word). Tanto uma como outra são directivas para o assemblador entender que o que se segue são dados e não instruções *Assembly* propriamente ditas.

Mas vamos então ver o programa que nos serviu de base a todas as explicações que demos até aqui. Poderá encontrar o seu código fonte na disqueta que acompanha a revista. Para assemblar e ligar faça respectivamente:

```
MASM relogio;
LINK relogio;
EXE2BIN relogio.exe relogio.com

ou

TASM relogio;
TLINK /t relogio;

ou ainda

A86 relogio.asm
```

Obterá um simpático programa que faz uma quantidade realmente fabulosa de trabalho em apenas 137 bytes de código máquina. É esta uma das maravilhas do *Assembly*.

E, como no próximo artigo vamos analisar os «misteriosos» ficheiros de estrutura .EXE, contamos consigo aqui, de pé firme.

José Páscoa