PROGRAMAR EM ASSEMBLY (PARTE III)

> **A função da memória nos PC é aproximadamente a mesma da memória nos humanos: guardar fielmente a informação**

No artigo anterior tivemos oportunidade de afirmar que o estudo da linguagem *Assembly* implica por si só a necessidade de ter um conhecimento algo (mais) pormenorizado de dois dos componentes deste sofisticado **sistema** que é o nosso computador pessoal compatível IBM PC.

Já abordámos um desses componentes, o **CPU** ou **microprocessador** e hoje iremos tratar de um outro, a **memória**.

Vamos introduzir ainda um outro tema de grande importância para a programação *Assembly* (e não só), os **interrupts** (ou **interrupções**) mas isso só após um ligeiro capítulo sobre *tabuada* **hexadecimal**.

1 – A MEMÓRIA

A função da memória nos PC é aproximadamente a mesma da memória nos humanos: guardar fielmente a informação. A memória dos PC porém, diga-se de passagem, é muito mais digna de confiança que a memória dos homens, pois é praticamente infalível. Ao contrário dos humanos que têm toda a sua memória no cérebro, os PC não localizam a totalidade (ou melhor, a quase totalidade) da sua memória no CPU mas sim em *chips* colocados, quer na placa principal (*motherboard*), quer em placas de expansão instaladas numa *slot*.

Já sabemos que para cumprir a delicada missão de guardar a informação, a memória dos PC é constituída por muitos milhares (ou milhões) de células de armazenamento (ou localizações) cada uma com 8 bits (1 byte) de tamanho. O valor de cada um dos bits do byte da célula permite que essa mesma célula assuma um entre 256 significados diferentes (2 elevado à oitava potência). Também cada uma dessas pequenas células é absolutamente individualizável, pois possui um **endereço absoluto** o qual se materializa por um valor numérico que identifica inequivocamente essa célula.

Mas o leitor talvez ainda esteja lembrado que os CPU dos computadores desta família têm uma *visão* segmentada da memória, isto é, não formam endereços absolutos, mas sim endereços relativos, os quais são constituídos por **pares de valores segmento\deslocamento**.

Contudo, a lógica interna de suporte ao sistema salva a situação e transforma directamente esses pares de valores em endereços absolutos quando for caso de aceder à memória. E, assim, a quantidade de memória *directamente endereçável* que pode ser instalada num sistema fica condicionada deste modo e, em última análise, ao número de Linhas de Sinal do Bus de Endereçamento.

Os «velhos» processadores 8088 e 8086 só têm 20 Linhas de Sinal e, por conseguinte, só podem endereçar 1 MB (2 elevado à vigésima potência), mas os processadores 80286 ou superiores podem endereçar pelo menos 16 MB e, por isso, os sistemas que deles dispõem já vêm hoje em dia quase sempre equipados à partida e no acto da compra com mais de 1 MB de memória, normalmente 2 MB ou 4 MB.

Mas o primeiro megabyte de memória apresenta a característica muito especial de poder ser endereçado pelo CPU quando este está a trabalhar em **modo real**, enquanto a memória acima de 1 MB só é endereçável e, por conseguinte, acedível pelo CPU a trabalhar em **modo protegido**.

Por esse motivo, a esse primeiro megabyte de memória dá-se muitas vezes o nome de **memória real** e à restante memória existentente acima desse primeiro megabyte dá-se o nome de **memória extendida**.

Punhamos agora aqui um importante parêntesis, para referir que pode existir ainda um terceiro tipo de memória (que gera muita confusão) e que dá correntemente pelos nomes de **memória expandida,** memória EMS (Expanded Memory Specification) ou memória LIM (Lotus/Intel/Microsoft). Este tipo de memória não é directamente endereçável pelo CPU; ao invés disso é constituída por blocos designados **páginas**, cada uma das quais normalmente com 16 KB de tamanho.

À informação guardada pelos programas em uma ou mais páginas da memória expandida é associada pelo *device driver gestor da memória expandida* um número designado por **handle**. As páginas de memória expandida associadas a um determinado *handle* são, por sua vez, numeradas a partir de zero recebendo o nome de **páginas lógicas.**

Os programas acedem à memória expandida informando o *device driver* sobre qual o *handle* e páginas lógicas que pretendem e este procede ao seu «mapeamento» numa zona normalmente de 64 KB da memória real que se designa por **page frame.**

A memória expandida pode ser obtida basicamente por três métodos:

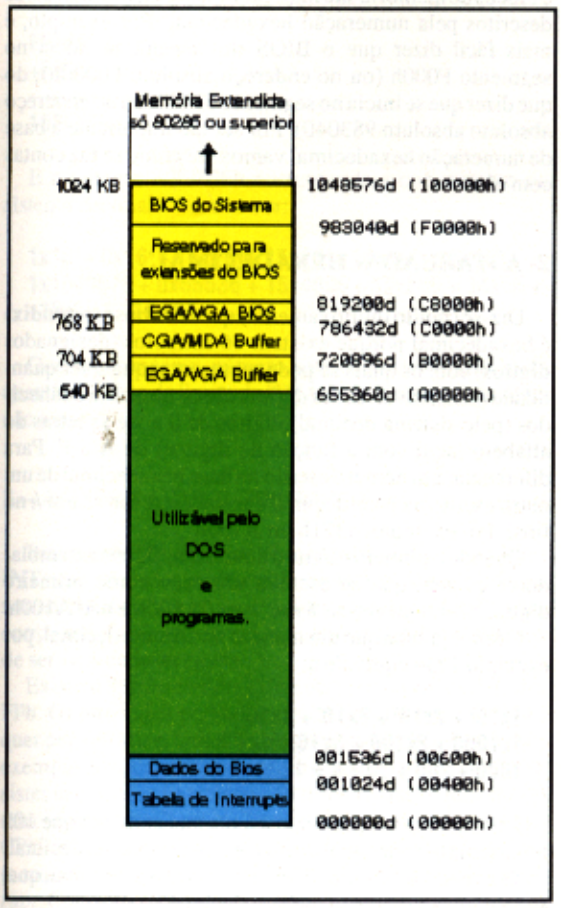
a) Através de placas adequadas instaladas numa *slot* de expansão ou algumas vezes *chips* integrados na placa principal. Este é o método clássico e é largamente utilizado nos computadores de processador 80286 (e alguns 8088/8086...). Para fazer o *interface* da placa com os programas é normalmente fornecido pelo fabricante um *device driver* adequado.

b) Por emulação e utilizando a memória extendida como base. É um procedimento também por *hardware* (ao nível do CPU) e só possível em computadores com processadores 80386 ou superiores e com o processador a trabalhar em modo virtual 86. O *device driver* EMM386.SYS (existem outros tais como o QEMM386.SYS e o 386MAX.SYS) fornecido com as versões recentes do MS-DOS faz aqui o *interface* entre os programas e o CPU.

c) Por simulação pura e simples (totalmente por *software*) utilizando memória extendida ou convencional como base. O conhecido *device driver* EMS40.SYS da PcMag é um bom exemplo do que se pode fazer mesmo quando não se tem o *hardware* adequado (e se está disposto a sacrifícios em termos de velocidade).

Mas esqueçamos este tipo especial de memória e debruce-mo-nos agora sobre a memória directamente en-

dereçável e com particular detalhe sobre a memória real. Repare na figura seguinte e nas três regiões coloridas respectivamente a azul,verde e amarelo:



As regiões azul e verde constituem aquilo que se designa por **memória convencional.** Nos computadores com processadores 8088 e 8086 a RAM é localizada apenas nesta zona, o que significa que esta poderá ter um máximo de 640 KB. Os computadores com processadores 80286 ou superiores podem, contudo, ter mais de 640 KB de RAM mas, nesse caso, o excesso estará localizado «fisicamente» a partir do endereço 1048576 (vd. a zona branca na figura) naquilo que se referiu atrás como sendo a **memória extendida.**

Os primeiros 64 KB menos 16 bytes de memória extendida recebem o nome de **HMA** (High Memory Area) e apresentam a propriedade interessante de poderem ser endereçados em modo real quando a 21ª linha de endereçamento (linha A20) é disponibilizada para uso nesse mesmo modo modo real. A faculdade de se poder endereçar a HMA em modo real só é possível devido à interpretação segmentada da memória (ora cá está uma vantagem, pensarão os leitores que estavam a ficar com a opinião que a segmentação da memória só parecia ter aspectos negativos...) pelos processadores desta família.

Imagine que um registo de segmento, por exemplo o registo ES, contém o valor hexadecimal FFFFh (ou a 15x4096+15x256+15x16+15x1 = 65535 em decimal). Imagine também que o registo SI contém o valor hexadecimal 10h (que corresponde em decimal a: 1x16+0x1 = 16).

Se se recorda ainda do que dissemos no artigo anterior, sabe que os registos de segmento percorrem a memória a «passadas» de 1 parágrafo de cada vez (cada parágrafo são 16 bytes) e os registos que nos dão o deslocamento percorrem essa mesma memória a «passadas» de apenas 1 byte de cada vez. Deste modo, uma instrução *Assembly* como por exemplo:

MOV AX, ES:[SI]

que informa o processador para **MOV**er para o registo AX o valor existente na célula de memória cujo endereço absoluto é dado pelo valor do registo de segmento ES vezes 16 (decimal) e somado do valor do registo SI, tem como resultado, no nosso caso, carregar para o registo AX o valor contido na célula de memória cujo endereço absoluto é 1,048,576 (65535x16 + 16).

Se reparar na figura anterior, verá que o endereço absoluto que foi obtido corresponde exactamente ao ínicio da memória extendida!

Pela mesma linha de raciocínio veríamos que o endereço maior que é possível de produzir pelo processador a trabalhar em modo real é FFFF:FFFF (esta é a convenção para se representar um endereço segmentado: segmento:deslocamento), isto é, os registos de segmento e deslocamento estarem carregados ambos com o valor FFFFh. Isto corresponde ao endereço absoluto:

65535x16+65535 = 1,114,095

A área de memória compreendida entre os endereços absolutos 1,048,576 e 1,114,095 (ou 100000h e 10FFEFh em hexadecimal) é precisamente a HMA. E, como se vê, o seu tamanho é precisamente 64 KB menos 16 bytes. Note que é condição para a HMA poder ser acedida em modo real, a linha A20 estar aberta (e esta é uma das missões do conhecido *device driver* HIMEM.SYS: abrir e fechar a HMA consoante as necessidades e as instruções que recebe do MS-DOS e outros programas), pois em caso contrário dá-se um *wrap around* (isto é, começamos de novo pelo endereço absoluto 00000h). A importância da HMA tem sido extraordinária nos tempos mais recentes, em particular pelo uso que o MS-DOS, Windows e alguns outros programas lhe têm dado. Mas regressemos à descrição das áreas de memória tomando como «pano de fundo» o desenho anterior.

A memória convencional é sempre limitada superiormente pelo endereço absoluto 655360 em decimal ou A0000h em hexadecimal, excepto quando o computador tem menos de 640 KB de RAM; nesse caso a memória convencional ficará reduzida na respectiva proporção. A região azul ocupa o início da memória convencional e é reservada por um programa residente no topo do ROM (chamado *Basic Input Output System* ou abreviadamente BIOS) para seu uso próprio e para uso pelo sistema operativo MS-DOS.

A região azul compreende a **Tabela de Interrupts** de que falaremos mais abaixo e a **Bios Data Area.**

Como os Bios (do Sistema e do Video) estão em ROM que é memória que só permite a escrita, surge a necessidade de guardar a informação temporária em algum sítio; esse sítio é precisamente a *Bios Data Area.*

Para além dos BIOS, ainda o DOS e a linguagem BASIC (porque é boa companheira dos PC *made by* IBM) têm direito a alguns endereços nessa região. Com tantos candidatos houve necessidade de reservar 512 bytes para guardar a informação temporária de todos eles.

Bem, para ser mais preciso, os PS/2 da IBM e *clones* muito semelhantes reservam ainda uma outra área em local não predeterminado (algumas vezes no último kilobyte da memória convencional) designada *Extended Bios Data Area.* Guarda-se na EBDA por exemplo informação temporária do *mouse.*

Muitos endereços da Bios Data Area têm excepcional interesse para o programador em linguagem *Assembly.* Contudo abstemo-nos de os transcrever, pois o espaço de revista que isso exigiria seria excessivo, e eles constam de qualquer bom livro sobre programação de PC.

Esperamos contudo dar nos próximos números um ou outro exemplo de programação que inclua informação

“A memória convencional é sempre limitada superiormente pelo endereço absoluto 655360 em decimal ou A0000h em hexadecimal”

recolhida nesses famosos endereços.

A região verde é ocupada parcialmente pelo sistema operativo MS-DOS (entre um mínimo actual de cerca de 12 KB até mais de 100 KB consoante a versão e a configuração, e também reconhecendo que uma parte do sistema operativo pode ser colocada na HMA nas versões mais recentes) e é também onde os nossos programas são normalmente carregados para serem executados. O sistema operativo MS-DOS é constituído, como se sabe, por três programas: IO.SYS, MSDOS.SYS e COMMAND.COM.

Acima da memória convencional e com início **sempre** em A0000H inicía-se o ROM (a amarelo na figura). Esta região, que totaliza 384 KB, é designada também por **Upper Memory** (de preferência não traduza pois existe alguma falta de uniformidade no significado das várias traduções existentes). Os primeiros 128 KB do ROM são reservados às placas gráficas de todos os tipos existentes para poderem apresentar ao sistema uma área que permita a escrita (a qual deixará nesta altura de ser ROM na verdadeira acepção da palavra). Essa área é muitas vezes designada por **buffer da placa gráfica** (veja-a como se fosse uma janela aberta pelo sistema para a **memória video** ou **display memory** existente na placa gráfica) e pode ocupar a região B0000H-B8000H (placas MDA e Hercules), a região B8000H-C0000H (placas CGA) ou eventualmente qualquer parte ou toda a região A0000H-C0000H (placas EGA, VGA e SuperVGA), tudo dependendo do modo gráfico emulado.

> **Acima da memória convencional e com início sempre em A0000H inicía-se o ROM**

A região seguinte do ROM e com início no endereço C0000H é normalmente ocupada pelo programa gestor do funcionamento das placas gráficas EGA, VGA e SuperVGA, o qual é designado por *Video Bios* dessas placas. O *Video Bios* ocupa uma área variável entre 24 KB e 32 KB. Ao *Video Bios* segue-se uma região muitas vezes desocupada excepto quando o sistema possui equipamentos periféricos que requeiram o **mapeamento** de programas nessa zona, os quais são designados **extensões do BIOS**.

Finalmente, a região de ROM com início em F0000H, é ocupada pelos vários programas que permitem o arranque do sistema e se ocupam, a partir daí, em facilitar um diálogo entre o *hardware* e o MS-DOS ou as nossas aplicações. Esse conjunto de programas é designado por **BIOS do Sistema**.

Todos os programas residentes em ROM ou mapeados nessa zona por equipamentos ou dispositivos são também genéricamente designados por **firmware,** para pôr em evidência o seu carácter de permanência. As áreas da Upper Memory não utilizadas pelo *firmware* podem, muitas vezes, ser aproveitadas para «mapeamento» de RAM existente na memória extendida ou expandida. Isso é possível com todos os computadores de processador 80386 ou superior e também com alguns 80286.

Após o mapeamento de RAM ficam constituídos **Upper Memory Blocks (UMB)** que podem ser aproveitados para carregamento de programas ou dados. Os UMB são criados e geridos com o auxílio de programas designado **gestores de UMB** (UMB *providers*), os mais conhecidos dos quais são o EMM386.SYS, QEMM, 386MAX e QRAM.

E é apenas isto o que de mais essencial há a saber sobre a memória.

Mas não queremos passar ao capítulo seguinte sem lhe chamar a atenção para um facto muito interessante.

O leitor deve já ter reparado que a base numérica hexadecimal se «encaixa às mil maravilhas» na estrutura segmentada de memória dos PC, e que é muito mais fácil descrever um endereço de memória em termos hexadecimais do que em termos decimais. Basicamente, isto tem a ver com o facto do tamanho em bytes de 1segmento de memória ser representado pelo valor 65536 em numeração decimal, mas por 10000 em numeração hexadecimal. E se fizer o favor de olhar mais uma vez para a figura, vai notar que os endereços absolutos dos ínicios das varias regiões e blocos de memória são mais facilmente memorizáveis se descritos pela numeração hexadecimal. Por exemplo, é mais fácil dizer que o BIOS do sistema se inicia no segmento F000h (ou no endereço absoluto F0000h), do que dizer que se inicia no segmento 61440 (ou no endereço absoluto absoluto 983040). Por ser tão importante a base de numeração hexadecimal vamos ver como se faz contas com ela.

2- A «TABUADA» HEXADECIMAL

Diz-se (muito intuitivamente) que uma **base** (ou **radix**) é hexadecimal porque existem 16 símbolos (designados **dígitos**) com os quais se pode representar qualquer quantidade mensurável. Esses dígitos são os nossos já conhecidos (pelo sistema decimal) dígitos de 0 a 9 e as letras do alfabeto (aqui com a função de dígitos) de A a F. Para diferenciar um número escrito na base hexadecimal de um outro escrito na base decimal é usual escrever-se um *h* no final. Por exemplo: 4321h ou A100h

Quando o primeiro dígito é uma **letra**, alguns assembladores exigem que se escreva um **zero** como primeiro dígito. Assim, dever-se-á escrever 0A100h e não A100h.

O leitor já sabe que um número no sistema decimal, por exemplo 1234 equivale a:

$$1x10^3 + 2x10^2 + 3x10^1 + 4x10^0 =$$
$$1x1000 + 2x100 + 3x10 + 4x1 =$$
$$1000 + 200 + 30 + 4 = 1234$$

Pois na base hexadecimal é a mesma coisa, só que 10h no sistema hexadecimal equivale a 16 no sistema decimal!

Se o número 1234h estivesse na base hexadecimal, qual seria o seu equivalente na base decimal?

Muito simplesmente:

$$1x16^3 + 2x16^2 + 3x16^1 + 4x16^0 =$$
$$1x4096 + 2x256 + 3x16 + 4 =$$
$$4096 + 512 + 48 + 4 = 4660$$

E no caso contrário, se tivéssemos o número 4660 em decimal e quiséssemos saber o seu equivalente em hexadecimal? Procedíamos a divisões sucessivas por 16:

4660/16 = 291 e resto 4.
291/16 = 18 e resto 3
18/16 = 1 e resto 2

E o número procurado é constituído pelo último quociente e pelos vários restos, isto é o número 1234h. E agora esta pequena questão:

Qual é o método prático de converter um endereço segmentado num endereço absoluto? Como se sabe os endereços segmentados vêm em princípio em formato hexadecimal e separados por 2 pontos, por exemplo:

1AB4:21A5

O método consiste em efectuar uma soma em que o valor do segmento é multiplicado implicitamente por 10h (um parágrafo) antes de ser adicionado ao deslocamento. Para o endereço segmentado acima indicado ficaria:

```
1AB40h
+21A5h
-----------
1CCE5h
```

Claro que neste exemplo não houve situações de «e vai um». Mas se houvesse, também não seria «nada do outro mundo». Por exemplo, para o endereço segmentado FFFF:FFFF teremos:

FFFF0h
+FFFFh
10FFEFh

E o número 10FFEFh corresponde a que número no sistema decimal? Vamos ver:

$1x16^5 + 0x16^4 + 15x16^3 + 15x16^2 + 14x16^1 + 15x16^0 =$
1x1048576 + 0x65536 + 15x4096 + 15x256 + 14x16 + +15x1= 1 114 095

Que foi o mesmo número que obtivemos mais atrás quando falámos do limite da HMA. E de facto a matemática não mente!

3 – A TABELA DE INTERRUPTS

Interrupts (designados em*assembly* pela mnemónica **INT**) são acções que determinam interrupções no fluxo normal de um programa e desviam a atenção do CPU para outros programas designados *interrupt handlers* (**rotinas de serviço de interrupts**).

Existem 256 *interrupts* diferentes e numerados de 00h a FFh. Os *interrupts* podem ser gerados quer por *hardware* quer por *software*. Quando são gerados por *hardware* (por exemplo por um periférico como o teclado, pelo relógio do sistema ou pelo controlador do disco rígido) é necessário um *chip* que coordene os *interrupts* já que não é possível ao CPU fazer executar dois *interrupt handlers* simultaneamente. Esse*chip* designa-se **PIC** *(Programmable Interrupt Controller)*. Quando os*interrupts* são gerados por *software*, é a própria instrução **INT** seguida de um número que provoca a iniciação de um *interrupt*.. Por exemplo: INT 20h. Os *interrupts* são interessantes porque representam um modo bastante económico de fazer executar um trabalho.

Por exemplo, introduz-se INT 20h no seio de um programa e o CPU sabe exactamente o que tem de fazer.

Quer o *interrupt* seja emitido por *hardware*, quer por *software*, o microprocessador apenas necessita conhecer o seu número para proceder sempre do seguinte modo:

1 – O conteúdo do registo *flags* é armazenado no *stack*. Logo, o valor do *stack pointer* (registo SP) é diminuido em 2.

2 – Os flags *Interrupt Flag* e *Trap Flag* são levados a 0. Recorde-se que no artigo anterior escrevemos que o *Interrupt Flag* em 0 impede que outra interrupção ocorra antes desta que se inicia ter sido tratada.

O *Trap Flag* a 0 impede a execução passo a passo do programa. Programas como o Debug do MS-DOS usam este *flag* para poderem efectuar a execução de programas ao «ralenti».

3 – O conteúdo do registo CS *(Code Segment)* é também armazenado no *stack*.

4 – O conteúdo do registo IP *(Instruction Pointer)* é tal como os anteriores copiado para o *stack*. Neste momento o valor de SP já diminuiu de 6 unidades em relação ao valor inicial.

5 – Multiplica o número do *interrupt* por 4 e o resultado obtido serve de índice na Tabela de Interrupts. Por exemplo, se o *interrupt* for o INT 10h, o microprocessador multiplica 10h por 4 e obtém 40h. Os quatro bytes que se iniciam no endereço 0000:0040 formam por sua vez um endereço onde se encontra o respectivo *interrupt handler*. A situação é um pouco semelhante a ir perguntar à Maria onde está o Manel.

Imagine agora que os bytes eram os seguintes:

0000:0040 12
0000:0041 .... A0
0000:0042 30
0000:0043 14

Então o endereço do *interrupt handler* é: 1430:A012
Exactamente este e não: 12A0:3014

Isto porque os PC armazenam em memória os números de trás para diante: um byte **mais significativo** encontra-se posicionado num endereço de memória mais elevado que um byte **menos significativo**. Fique alertado, contudo, para o facto de que nem todos os computadores assim procedem, mas também não é um «mal exclusivo» dos PC.

6 – Então (e aproveitando o endereço que obtivemos no ponto 4), o CPU coloca os 2 primeiros bytes no registo IP, logo IP ficará com o valor 0A012h.

Coloca os dois seguinte, 1430h no registo CS.

7 – Após essa operação a execução inicia-se precisamente no endereço 1430:A012, que é o ínicio do *interrupt handler*.

8 – No final do *interrupt handler* existe uma instrução representada em *Assembly* por **IRET** (Interrupt Return). A execução dessa instrução determina a inversão do processo, designadamente,

9 – O conteúdo do «velho» IP é retirado do *stack* e restaurado nesse registo.

10 – O conteúdo do «velho» CS idem aspas.

11 – Os *flags* também são restaurados.

12 – E muito naturalmente a execução prossegue como se nada se tivesse passado.

Mas, na realidade, muita coisa mesmo se pode ter passado naqueles milisegundos que durou a execução desse *interrupt handler*. Muito, muito mais há a dizer sobre os *interrupts*, por isso reservamos o nosso próximo artigo exactamente para desenvolver o tema e fazer tentativamente um pequeno programa que lide com os ditos cujos *interrupts*. E vai ver que muito em breve está a programar em *Assembly* sem dar por isso, porque a linguagem *Assembly* é efectivamente muito simples, o que é difícil é... a *entourage*.

“ Interrupts (designados em *assembly* pela mnemónica INT) são acções que determinam interrupções no fluxo normal de um programa e desviam a atenção do CPU para outros programas ”

José Páscoa