```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 01 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
i------------©
¦ ASSEMBLY I ¦
È------------¥
```

A linguagem ASSEMBLY (e não assemblER!) dá medo em muita gente! Só não sei porque! As linguagens ditas de "alto nível" são MUITO mais complexas que o assembly! O programador assembly tem que saber, antes de mais nada, como está organizada a memória da máquina em que trabalha, a disponibilidade de rotinas pré-definidas na ROM do micro (que facilita muito a vida de vez em quando!) e os demais recursos que a máquina oferece.

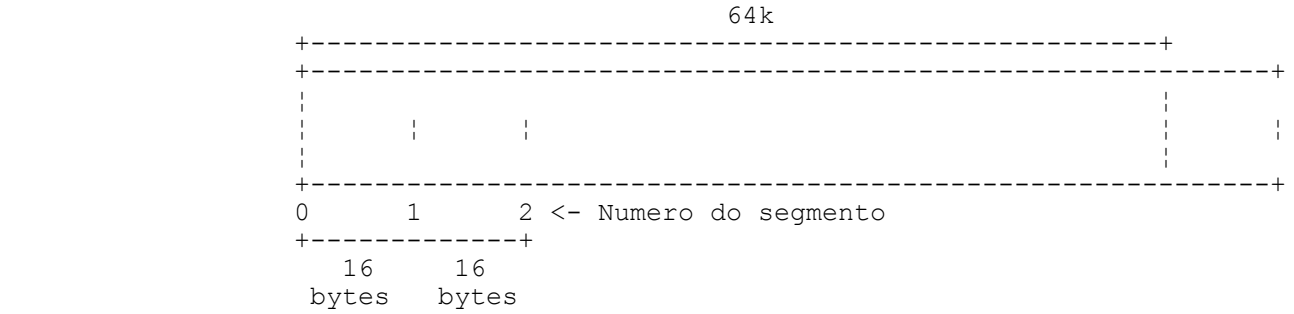
Uma grande desvantagem do assembly com relação as outras linguagens é que não existe tipagem de dados como, por exemplo, ponto-flutuante... O programador terá que desenvolver as suas próprias rotinas ou lançar mão do co-processador matemático (o TURBO ASSEMBLER, da Borland, fornece uma maneira de emular o co-processador). Não existem funções de entrada-saída como PRINT do BASIC ou o Write() do PASCAL... Não existem rotinas que imprimam dados numéricos ou strings na tela... Enfim... não existe nada de útil! (Será?! hehehe)

Pra que serve o assembly então? A resposta é: Para que você possa desenvolver as suas próprias rotinas, sem ter que topar com bugs ou limitações de rotinas já existentes na ROM-BIOS ou no seu compilador "C", "PASCAL" ou qualquer outro... Cabe aqui uma consideração interessante: É muito mais produtivo usarmos uma linguagem de alto nível juntamente com nossas rotinas em assembly... Evita-se a "reinvenção da roda" e não temos que desenvolver TODAS as rotinas necessárias para os nossos programas. Em particular, o assembly é muito útil quando queremos criar rotinas que não existem na linguagem de alto-nível nativa! Uma rotina ASM bem desenvolvida pode nos dar a vantagem da velocidade ou do tamanho mais reduzido em nossos programas.

O primeiro passo para começar a entender alguma coisa de assembly é entender como a CPU organiza a memória. Como no nosso caso a idéia é entender os microprocessadores da família 80x86 da Intel (presentes em qualquer PC-Compatível), vamos dar uma olhadela no modelamento de memória usado pelos PCs, funcionando sob o MS-DOS (Windows, OS/2, UNIX, etc... usam outro tipo de modelamento... MUITO MAIS COMPLICADO!).

```
i-----------------------------------------©
¦ Modelamento REAL da memória - A segmentação ¦
È-----------------------------------------¥
```

A memória de qualquer PC é dividida em segmentos. Cada segmento tem 64k bytes de tamanho (65536 bytes) e por mais estranho que pareça os segmentos não são organizados de forma seqüencial (o segmento seguinte não começa logo após o anterior!). Existe uma sobreposição. De uma olhada:



O segundo segmento começa exatamente 16 bytes depois do primeiro. Deu pra perceber que o inicio do segundo segmento está DENTRO do primeiro, já que os segmentos tem 64k de tamanho!

Este esquema biruta confunde bastante os programadores menos experientes e, até hoje, ninguém sabe porque a Intel resolveu utilizar essa coisa esquisita. Mas, paciência, é assim que a coisa funciona!

Para encontrarmos um determinado byte dentro de um segmento precisamos fornecer o OFFSET (deslocamento, em inglês) deste byte relativo ao inicio do segmento. Assim, se queremos localizar o décimo-quinto byte do segmento 0, basta especificar 0:15, ou seja, segmento 0 e offset 15. Esta notação é usada no restante deste e de outros artigos.

Na realidade a CPU faz o seguinte cálculo para encontrar o "endereço físico" ou "endereço efetivo" na memória:

```
+-----------------------------------------------------------------+
¦         ENDEREÇO-EFETIVO = (SEGMENTO * 16) + OFFSET             ¦
+-----------------------------------------------------------------+
```

Ilustrando a complexidade deste esquema de endereçamento, podemos provar que existem diversas formas de especificarmos um único "endereço efetivo" da memória... Por exemplo, o endereço 0:13Ah pode ser também escrito como:

0001h:012Ah	0002h:011Ah	0003h:010Ah	0004h:00FAh
0005h:00EAh	0006h:00DAh	0007h:00CAh	0008h:00BAh
0009h:00AAh	000Ah:009Ah	000Bh:008Ah	000Ch:007Ah
000Dh:006Ah	000Eh:005Ah	000Fh:004Ah	0010h:003Ah
0011h:002Ah	0012h:001Ah	0013h:000Ah	

Basta fazer as contas que você verá que todas estas formas darão o mesmo resultado: o endereço-efetivo 0013Ah. Generalizando, existem, no máximo, 16 formas de especificarmos o mesmo endereço físico! As únicas faixas de endereços que não tem equivalentes e só podem ser especificados de uma única forma são os dezesseis primeiros bytes do segmento 0 e os últimos dezesseis bytes do segmento 0FFFFh.

Normalmente o programador não tem que se preocupar com esse tipo de coisa. O compilador toma conta da melhor forma de endereçamento. Mas, como a toda regra existe uma excessão, a informação acima pode ser útil algum dia.

```
+-----------------------------------------------------------------+
¦ A BASE NUMÉRICA HEXADECIMAL E BINARIA (para os novatos...)      ¦
+-----------------------------------------------------------------+
```

Alguns talvez não tenham conhecimento sobre as demais bases

numéricas usadas na área informata. É muito comum dizermos "código hexadecimal", mas o que significa?

É bastante lógico que usemos o sistema decimal como base para todos os cálculos matemáticos do dia-a-dia pelo simples fato de temos DEZ dedos nas mãos... fica fácil contar nos dedos quando precisamos (hehe).

Computadores usam o sistema binário por um outro motivo simples: Existem apenas dois níveis de tensão presentes em todos os circuitos lógicos: níveis baixo e alto (que são chamados de 0 e 1 por conveniência... para podermos medi-los sem ter que recorrer a um multímetro!). O sistema hexadecimal também tem o seu lugar: é a forma mais abreviada de escrever um conjunto de bits.

Em decimal, o númeroo 1994, por exemplo, pode ser escrito como:

```
1994 = (1 * 10^3) + (9 * 10^2) + (9 * 10^1) + (4 * 10^0)
```

Note a base 10 nas potências. Faço agora uma pergunta: Como representaríamos o mesmo númeroo se tivéssemos 16 dedos nas mãos?

¦ Primeiro teríamos que obter mais dígitos... 0 até 9 não são suficientes. Pegaremos mais 6 letras do alfabeto para suprir esta deficiência.

¦ Segundo, Tomemos como inspiração um odômetro (equipamento disponível em qualquer automóvel - é o medidor de quilometragem!): Quando o algarismo mais a direita (o menos significativo) chega a 9 e é incrementado, o que ocorre?... Retorna a 0 e o próximo é incrementado, formando o 10. No caso do sistema hexadecimal, isto só acontece quando o último algarismo alcança F e é incrementado! Depois do 9 vem o A, depois o B, depois o C, e assim por diante... até chegar a vez do F e saltar para 0, incrementando o próximo algarismo, certo?

Como contar em base diferente de dez é uma situação não muito intuitiva, vejamos a regra de conversão de bases. Começaremos pela base decimal para a hexadecimal. Tomemos o númeroo 1994 como exemplo. A regra é simples: Divide-se 1994 por 16 (base hexadecimal) até que o quoeficiente seja zero... toma-se os restos e tem-se o númeroo convertido para hexadecimal:

```
+-------------------------------------------------------------+
¦ 1994 / 16     -> Q=124, R=10      -> 10=A                   ¦
¦ 124 / 16      -> Q=7, R=12        -> 12=C                   ¦
¦ 7 / 16        -> Q=0, R=7         ->  7=7                   ¦
+-------------------------------------------------------------+
```

Toma-se então os restos de baixo para cima, formando o númeroo em hexadecimal. Neste caso, 1994=7CAh

Acrescente um 'h' no fim do númeroo para sabermos que se trata da base 16, do contrário, se olharmos um númeroo "7CA" poderíamos associa-lo a qualquer outra base numérica (base octadecimal por exemplo!)...

O processo inverso, hexa->decimal, é mais simples... basta escrever o número, multiplicando cada digito pela potência correta, levando-se em conta a equivalencia das letras com a base decimal:

```
+-------------------------------------------------------------------+
|  7CAh = (7 * 16^2) + (C * 16^1) + (A * 16^0) =                    |
|         (7 * 16^2) + (12 * 16^1) + (10 * 16^0) =                  |
|         1792 + 192 + 10 = 1994                                    |
+-------------------------------------------------------------------+
```

As mesmas regras podem ser aplicadas para a base binária (que tem apenas dois digitos: 0 e 1). Por exemplo, o númeroo 12 em binário fica:

```
+-------------------------------------------------------------------+
|  12 / 2      -> Q=6, R=0                                          |
|  6 / 2       -> Q=3, R=0                                          |
|  3 / 2       -> Q=1, R=1                                          |
|  1 / 2       -> Q=0, R=1                                          |
|                                                                   |
|  12 = 1100b                                                       |
+-------------------------------------------------------------------+
```

Cada digito na base binária é conhecido como BIT (Binary digIT - ou digito binário, em inglês)! Note o 'b' no fim do númeroo convertido...

Faça o processo inverso... Converta 10100110b para decimal.

A vantagem de usarmos um númeroo em base hexadecimal é que cada digito hexadecimal equivale a exatamente quatro digitos binários! Faça as contas: Quatro bits podem conter apenas 16 números (de 0 a 15), que é exatamente a quantidade de digitos na base hexadecimal.

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 02 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
i-------------©
¦ ASSEMBLY II ¦
È-------------¥
```

Mais alguns conceitos são necessários para que o pretenso programador ASSEMBLY saiba o que está fazendo. Em eletrônica digital estuda-se a algebra booleana e aritimética com números binários. Aqui esses conceitos também são importantes... Vamos começar pela aritimética binária:

A primeira operação básica - a soma - não tem muitos mistérios... basta recorrer ao equivalente decimal. Quando somamos dois números decimais, efetuamos a soma de cada algarismo em separado, prestando atenção aos "vai um" que ocorrem entre um algarismo e outro. Em binário fazemos o mesmo:

```
+-------------------------------------------------------------------+
¦ 1010b + 0110b = ?                                                 ¦
¦                                                                   ¦
¦    111         <- "Vai uns"                                       ¦
¦     1010b                                                         ¦
¦   + 0110b                                                         ¦
¦   ---------                                                       ¦
¦    10000b                                                         ¦
+-------------------------------------------------------------------+
```

Ora, na base decimal, quando se soma - por exemplo - 9 e 2, fica 1 e "vai um"... Tomemos o exemplo do odômetro (aquele indicador de quilometragem do carro!): 09 -> 10 -> 11

Enquanto na base decimal existem 10 algarismos (0 até 9), na base binária temos 2 (0 e 1). O odômetro ficaria assim:
00b -> 01b -> 10b -> 11b

Portanto, 1b + 1b = 10b ou, ainda, 0b e "vai um".

A subtração é mais complicada de entender... Na base decimal existem os números negativos... em binário nao! (Veremos depois como "representar" um númeroo negativo em binário!). Assim, 1b - 1b = 0b (lógico), 1b - 0b = 1b (outra vez, evidente!), 0b - 0b = 0b (hehe... você deve estar achando que eu estou te sacaneando, né?), mas e 0b - 1b = ?????

A solução é a seguinte: Na base decimal quando subtraimos um algarismo menor de outro maior costumamos "tomar um emprestado" para que a conta fique correta. Em binário a coisa funciona do mesmo jeito, mas se não tivermos de onde "tomar um emprestado" devemos indicar que foi tomado um de qualquer forma:

```
+-----------------------------------------------------------------+
| 0b - 1b = ?                                                     |
|                                                                 |
|     1         <- Tomamos esse um emprestado de algum lugar!     |
|      0b                             (não importa de onde!)      |
|   -  1b                                                         |
|   ------                                                        |
|      1b                                                         |
+-----------------------------------------------------------------+
```

Esse "1" que apareceu por mágica é conhecido como BORROW. Em um númeroo binário maior basta usar o mesmo artificio:

```
+-----------------------------------------------------------------+
| 1010b - 0101b = ?                                               |
|                                                                 |
|      1 1         <- Os "1"s que foram tomados emprestados são   |
|      1010b          subtraídos no proximo digito.               |
|    - 0101b                                                      |
|   ---------                                                     |
|      0101b                                                      |
+-----------------------------------------------------------------+
```

Faça a conta: 0000b - 0001b, vai acontecer uma coisa interessante! Faça a mesma conta usando um programa, ou calculadora cientifica, que manipule números binários... O resultado vai ser ligairamente diferente por causa da limitação dos digitos suportados pelo software (ou calculadora). Deixo a conclusão do "por que" desta diferença para você... (Uma dica, faça a conta com os "n" digitos suportados pela calculadora e terá a explicação!).

```
i-----------------------------------------------------------------©
¦ Representando números negativos em binário                     ¦
È-----------------------------------------------------------------¥
```

Um artificio da algebra booleana para representar um númeroo interiro negativo é usar o último bit como indicador do sinal do númeroo. Mas, esse artificio gera uma segunda complicação...

Limitemos esse estudo ao tamanho de um byte (8 bits)... Se o bit 7 (a contagem começa pelo bit 0 - mais a direita) for 0 o númeroo representado é positivo, se for 1, é negativo. Essa é a diferença entre um "char" e um "unsigned char" na linguagem C - ou um "char" e um "byte" em PASCAL (Note que um "unsigned char" pode variar de 0 até 255 - 00000000b até 11111111b - e um "signed char" pode variar de -128 até 127 - exatamenta a mesma faixa, porém um tem sinal e o outro não!).

A complicação que falei acima é com relação à representação dos números negativos. Quando um númeroo não é nagativo, basta convertê-lo para base decimal que você saberá qual é esse númeroo, no entanto, números negativos precisam ser "complementados" para que saibamos o númeroo que está sendo representado. A coisa NãO funciona da seguinte forma:

```
+-----------------------------------------------------------------+
|  00001010b   =   10                                             |
|  10001010b   =  -10     (ERRADO)                                |
+-----------------------------------------------------------------+
```

Não basta "esquecermos" o bit 7 e lermos o restante do byte. O procedimento correto para sabermos que númeroo está sendo representado negativamente no segundo exemplo é:

¦ Inverte-se todos os bits
¦ Soma-se 1 ao resultado

```
+-----------------------------------------------------------------+
¦ 10001010b   ->  01110101b + 00000001b   ->  01110110b          ¦
¦ 01110110b   =   118                                            ¦
¦ Logo:                                                          ¦
¦ 10001010b   =  -118                                            ¦
+-----------------------------------------------------------------+
```

Com isso podemos explicar a diferença entre os extremos da faixa de um "signed char":

¦ Os números positivos contam de 00000000b até 01111111b, isto é, de 0 até 127.
¦ Os números negativos contam de 10000000b até 11111111b, isto é, de -128 até -1.

Em "C" (ou PASCAL), a mesma lógica pode ser aplicada aos "int" e "long" (ou INTEGER e LONGINT), só que a quantidade de bits será maior ("int" tem 16 bits de tamanho e "long" tem 32).

Não se preocupe MUITO com a representação de números negativos em binário... A CPU toma conta de tudo isso sozinha... mas, as vezes, você tem que saber que resultado poderá ser obtido de uma operação aritimética em seus programas, ok?

As outras duas operações matemáticas básicas (multiplicação e divisão) tanbém estão presentes nos processadores 80x86... Mas, não necessitamos ver como o processo é feito a nível binário. Confie na CPU! :)

```
i----------------------------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 03 ¦
È----------------------------------------------¥
```

Por: Frederico Pissarra

```
i--------------©
¦ ASSEMBLY III ¦
È--------------¥
```

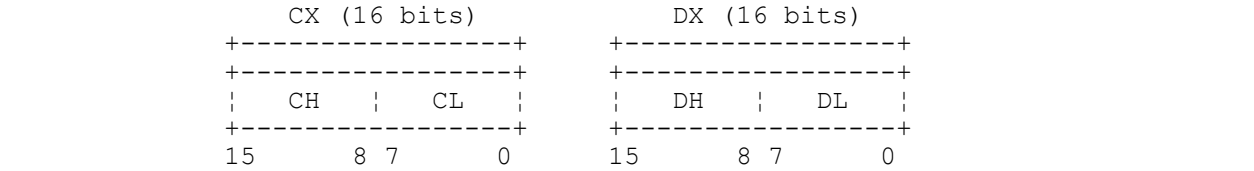
Comecemos a dar uma olhadela na arquitetura dos microprocessadores da família INTEL 80x86... Vamos aos registradores!

Entenda os registradores como se fossem variáveis que o microprocessador disponibiliza ao sistema. TODOS os registradores têm 16 bits de tamanho e aqui vai a descrição deles:

```
 +------+
 ¦  AX  ¦<--+
 +------¦   ¦
 ¦  BX  ¦<--¦
 +------¦   +- Registradores de uso geral
 ¦  CX  ¦<--¦
 +------¦   ¦
 ¦  DX  ¦<--+
 +------+
 +------+
 ¦  SI  ¦<---- índice FONTE (Source Index)
 +------¦
 ¦  DI  ¦<---- índice DESTINO (Destination Index)
 +------+
 +------+
 ¦  SP  ¦<---- Apontador de pilha (Stack Pointer)
 +------¦
 ¦  BP  ¦<---- Apontador de base (Base Pointer)
 +------+
 +------+
 ¦  CS  ¦<---- Segmento de Cógido (Code Segment)
 +------¦
 ¦  DS  ¦<---- Segmento de Dados (Data Segment)
 +------¦
 ¦  ES  ¦<---- Segmento de dados Extra (Extra data Segment)
 +------¦
 ¦  SS  ¦<---- Segmento de Pilha (Stack Segment)
 +------+
 +------+
 ¦  IP  ¦<---- Apontador de instrução (Instruction Pointer)
 +------+
 +------+
 ¦Flags ¦<---- Sinalizadores
 +------+
```

Por enquanto vamos nos deter na descrição dos registradores uso geral... Eles podem ser subdivididos em dois registradore de oito bits cada:

```
    AX (16 bits)           BX (16 bits)
+-----------------+    +-----------------+
+-----------------+    +-----------------+
¦   AH   ¦   AL   ¦    ¦   BH   ¦   BL   ¦
+-----------------+    +-----------------+
15      8 7      0     15      8 7      0
```



AH é o byte mais significativo do registrador AX, enquanto que AL é o menos significativo. Se alterarmos o conteúdo de AL, estaremos alterando o byte menos significativo de AX ao mesmo tempo... Não existem registradores de oito bits em separado... tudo é uma coisa só. Portanto, ao manipularmos AH, estaremos manipulando AX ao mesmo tempo!

O nome de cada registrador tem o seu sentido de ser... "A" de AX quer dizer que este registrador é um "acumulador" (usado por default em algumas operações matematicas!), por exemplo...

AX -> Acumulador
BX -> Base
CX -> Contador
DX -> Dados

O "X" de AX significa "eXtended". "H" de AH significa "High byte".

Embora estes registradores possam ser usados sem restrições, é interessante atribuir uma função para cada um deles nos nossos programas sempre que possível... Isto facilita a leitura do código e nos educa a seguirmos uma linha de raciocínio mais concisa... Mas, se for de sua preferência não seguir qualquer padrão no uso desses registradores, não se preocupe... não haverá qualquer desvantagem nisso (Well... depende do código, as vezes somos obrigados a usar determinado registrador!).

Alguns pontos importantes quanto a esses nomes serão observados no decorrer do curso... Por exemplo, certas instruções usam AX (ou AL, ou AH) e somente ele, não permitindo o uso de nenhum outro registrador... Outras, usam CX para contar, etc... essas instruções específicas serão vistas em outra oportunidade.

Os registradores SI e DI são usados como índices para tabelas. Em particular, SI é usado para leitura de uma tabela e DI para escrita (fonte e destino... lembra algum procedimento de cópia, nao?). No entanto, esses registradores podem ser usados com outras finalidades... Podemos incluí-los no grupo de "registradores de uso geral", mas assim como alguns registradores de uso geral, eles têm aplicação exclusiva em algumas instruções, SI e DI são usados especificamente como índices em instruções que manipulam blocos (também veremos isso mais tarde!).

Os registradores CS, DS, ES e SS armazenam os segmentos onde estão o código (programa sendo executado), os dados, os dados extras, e a pilha, respectivamente. Lembre-se que a memória é segmentada em blocos de 64kbytes (dê uma olhada na primeira mensagem dessa série).

Quando nos referimos, através de alguma instrução, a um endereço de memória, estaremos nos referindo ao OFFSET dentro de um segmento. O registrador de segmento usado para localizar o dado no offset especificado vai depender da própria instrução... Um exemplo em assembly:

```
+-----------------------------------------------------------------+
¦      MOV     AL,[1D4Ah]                                         ¦
+-----------------------------------------------------------------+
```

O númeroo hexadecimal entre os colchetes é a indicação de um offset em um segmento... Por default, a maioria das instruções usa o segmento de dados (valor em DS). A instrução acima é equivalente a:

```
+-----------------------------------------------------------------+
¦      AL = DS:[1D4Ah]                                            ¦
+-----------------------------------------------------------------+
```

Isto é, em AL será colocado o byte que está armazenado no offset 1D4Ah do segmento de dados (valor em DS). Veremos mais sobre os segmentos e as instruções mais tarde :)

Se quisessemos localizar o byte desejado em outro segmento (mas no mesmo offset) devemos especificar o registrador de segmento na instrução:

```
+-----------------------------------------------------------------+
¦      MOV     AL,ES:[1D4Ah]                                      ¦
+-----------------------------------------------------------------+
```

Aqui o valor de ES será usado.

O registrador IP (Instruction Pointer) é o offset do segmento de código que contém a próxima instrução a ser execuatda. Este registrador não é acessível por qualquer instrução (pelo menos não pelas documentadas pela Intel)... é de uso interno do microprocessador. No entanto existem alguns macetes para conseguirmos obter o seu conteúdo (o que na maioria das aplicações não é necessario... Para que conhecer o endereço da próxima instrução se ela var ser executada de qualquer jeito?).

O registrador SP é o offset do segmento SS (segmento de pilha) onde o próximo dado vai ser empilhado. A pilha serve para armazenar dados que posteriormente podem ser recuperados sem que tenhamos que usar um dos registradores para esse fim. Também é usada para armazenar o endereço de retorno das sub-rotinas. A pilha "cresce" de cima para baixo, isto é, SP é decrementado cada vez que um novo dado é colocado na pilha. Note também que existe um registrador de segmento exclusivo para a pilha... SP sempre está relacionado a esse segmento (SS), como foi dito antes.

Para ilustrar o funcionamento da pilha, no gráfico abaixo simularemos o empilhamento do conteúdo do registrador AX através da instrução:

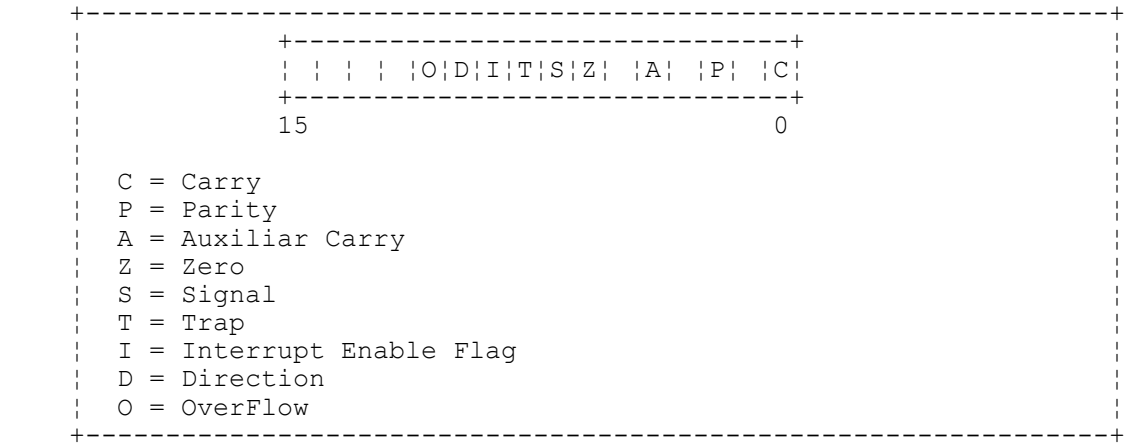
```
+-----------------------------------------------------------------+
¦      PUSH    AX                                                 ¦
+-----------------------------------------------------------------+

+-----------------------------------------------------------------+
¦  AX = A527h (Valor em AX)                                        ¦
¦                                                                 ¦
¦    +-------+                        +-------+                   ¦
¦    ¦ ????h ¦<----- SP = n           ¦ A527h ¦                   ¦
¦    +-------¦                        +-------¦                   ¦
¦    ¦       ¦                        ¦       ¦<----- SP = n - 1  ¦
¦    +-------+                        +-------+                   ¦
¦                                                                 ¦
¦ (antes de PUSH AX)                (depois de PUSH AX)           ¦
+-----------------------------------------------------------------+
```

Observe que SP sempre aponta para um espaço vago na pilha.

Na realidade SP é decrementado de duas posições ao invés de apenas uma... mas, esse detalhe deixo para mais tarde.

O registrador BP pode ser usado como apontador para a base da pilha (já que, por default, está relacionado a SS) ou como um registrador de uso geral... depende do seu programa. Veremos isso detalhadamente mais tarde.

Um dos registradores mais importantes de qualquer microprocessador é o de "Flags". Eis uma descrição dos bits deste registrador (a descrição abaixo aplica-se ao 8086. Normalmente não acessamos diretamente o registrador de flags - embora possamos fazê-lo - por isso não é conveniente assumirmos que os bits estão sempre no mesmo lugar para qualquer microprocessador da família 80x86!):



¦ Carry:

Esse flag é setado sempre quando houver "vai um" depois de uma adição ou quando há BORROW depois de uma subtração. Ou quando, numa operação de deslocamento de bits, o bit mais ao extremo for deslocado para fora do dado (suponha um byte... se todos os bits forem deslocados em uma posição para a direita, o que acontece com o bit 0?... Resposta: Vai para o carry!)

¦ Parity:

Depois de uma instrução aritimética ou lógica este bit informa se o resultado tem um númeroo par de "1"s ou não.

¦ Auxiliar Carry:

Igual ao carry, mas indica o "vai um" no meio de um dado (no caso de um byte, se houve "vai um" do bit 3 para o bit 4!).

¦ Zero:

Depois de uma operação aritimética ou lógica, esse flag indica se o resultado é zero ou não.

¦ Signal:

Depois de uma instrução aritimética ou lógica, este flag é uma cópia do bit de mais alta ordem do resultado, isto é, seu

sinal (dê uma olhada na "representação de números negativos em binário" no texto anterior!).

¦ Trap:

Quando setado (1) executa instruções passo-a-passo... Não nos interessa estudar esse bit por causa das diferenças de implementação deste flag em toda a família 80x86.

¦ Interrupt Enable Flag

Habilita/Desabilita o reconhecimento de interrupções mascaráveis pela CPU. Sobre interrupções, veremos mais tarde!

¦ Direction:

Quando usamos instruções de manipulação de blocos, precisamos especificar a direção que usaremos (do inicio para o fim ou do fim para o inicio).
Quando D=0 a direção é a do início para o fim... D=1, então a direção é contrária!

¦ OverFlow:

Depois de uma instrução aritimética ou lógica, este bit indica se houve mudança no bit mais significativo, ou seja, no sinal. Por exemplo, se somarmos FFFFh + 0001h obteremos 00h. O bit mais significativo variou de 1 para 0 (o counteúdo inicial de um registrador era FFFFh e depois da soma foi para 0000h), indicando que o resultado saiu da faixa (overflow) - ora, FFFFh + 0001h = 10000h, porém um registrador tem 16 bits de tamanho e o resultado cabe em 17 bits. Neste exemplo, o bit de carry também será setado pois houve "vai um" do bit 15 para o inexistente bit 16, mas não confunda o flag de overflow com o carry!

Quando aos demais bits, não se pode prever seus estados lógicos (1 ou 0).

Na próxima mensagem começaremos a ver algumas instruções do microprocessador 8086. Ainda não escreveremos nenhum programa, a intenção é familiarizá-lo com a arquitetura do microprocessador antes de começarmos a colocar a mão na massa... tenha um pouco de paciência! :)

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 04 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
i-------------©
¦ ASSEMBLY IV ¦
È-------------¥
```

Começaremos a ver algumas instruções do microprocessador 8086 agora. Existem os seguintes tipos de instruções:

¦ Instruções Aritiméticas
¦ Instruções Lógicas
¦ Instruções de Controle de Fluxo de Programa
¦ Instruções de manipulação de flags
¦ Instruções de manipulação da pilha
¦ Instruções de manipulação de blocos
¦ Instruções de manipulação de registradores/dados
¦ Instruções de Entrada/Saída

Vamos começar com as instruções de manipulação de registradores/dados por serem estas as mais fáceis de entender.

```
i---------------©
¦ Instrução MOV ¦
È---------------¥
```

MOV tem a finalidade de MOVimentar um dado de um lugar para outro. Por exemplo, para carregar um registrador com um determinado valor. Isto é feito com MOV:

```
+-----------------------------------------------------------------+
¦  MOV AX,0001h                                                   ¦
+-----------------------------------------------------------------+
```

É a mesma coisa que dizer: "AX = 1". Na verdade, movimentamos o valor 1 para dentro do registrador AX.

Podemos mover o conteúdo de um registrador para outro:

```
+-----------------------------------------------------------------+
¦  MOV BH,CL                                                      ¦
+-----------------------------------------------------------------+
```

É a mesma coisa que "BH = CL"!

Os registradores de segmento não podem ser inicializados com MOV tomando um parametro imediato (numérico). Esses registradores são inicializados indiretamente:

```
+-----------------------------------------------------------------+
¦  MOV DS,0    ; ERRADO!!!                                        ¦
¦                                                                 ¦
¦  MOV AX,0                                                       ¦
¦  MOV DS,AX   ; CORRETO!                                         ¦
+-----------------------------------------------------------------+
```

Carregar um registrador com o conteúdo (byte ou word, depende da instrução!) armazenado em um segmento é simples, basta especificar o offset do dado entre colchetes. Atenção que o segmento de dados (DS) é assumido por default com algumas excessões:

```
MOV AL,[0FFFFh]
```

A instrução acima, pega o byte armazenado no endereço DS:FFFFh e coloca-o em AL. Sabemos que um byte vai ser lido do offset especificado porque AL tem 8 bits de tamanho.

Ao invés de usarmos um offset imediato podemos usar um registrador:

```
MOV BX,0FFFFh
MOV CH,[BX]
```

Neste caso, BX contém o offset e o byte no endereço DS:BX é armazenado em CH. Note que o registrador usado como indice obrigatoriamente deve ser de 16 bits.

Uma observação quanto a essa modalidade: Dependendo do registrador usado como offset, o segmento default poderá ser DS ou SS. Se ao invés de BX usassemos BP, o segmento default seria SS e não DS - de uma olhada no diagrama de distribuição dos registradores no texto anterior. BP foi colocado no mesmo bloco de SP, indicando que ambos estão relacionados com SS (Segmento de pilha) - Eis uma tabela das modalidades e dos segmentos default que podem ser usados como offset:

Offset usando registros	Segmento default
[SI + deslocamento]	DS
[DI + deslocamento]	DS
[BP + deslocamento]	SS
[BX + deslocamento]	DS
[BX + SI + deslocamento]	DS
[BX + DI + deslocamento]	DS
[BP + SI + deslocamento]	SS
[BP + DI + deslocamento]	SS

O "deslocamento" pode ser suprimido se for 0.

Você pode evitar o segmento default explicitando um registrador de segmento na instrução:

```
MOV DH,ES:[BX]      ;Usa ES ao invés de DS
MOV AL,CS:[SI + 4]  ;Usa CS ao invés de DS
```

Repare que tenho usado os registradores de 8 bits para armazenar os dados... Pode-se usar os de 16 bits também:

```
MOV ES:[BX],AX         ; Poe o valor de AX para ES:BX
```

Só que neste caso serão armazenados 2 bytes no endereço ES:BX. O primeiro byte é o menos significativo e o segundo o mais signigicativo. Essa instrução equivale-se a:

```
 +-----------------------------------------------------------------+
 ¦  MOV ES:[BX],AL            ; Instruçõess que fazem a mesma      ¦
 ¦  MOV ES:[BX + 1],AH        ;coisa que MOV ES:[BX],AX            ¦
 +-----------------------------------------------------------------+
```

Repare também que não é possível mover o conteúdo de uma posição da memória para outra, diretamente, usando MOV. Existe outra instrução que faz isso: MOVSB ou MOVSW. Veremos essas instruções mais tarde.

Regra geral: Um dos operandos TEM que ser um registrador! Salvo no caso da movimentação de um imediato para uma posição de memória:

```
 +-----------------------------------------------------------------+
 ¦  MOV [DI],[SI]       ; ERRO!                                   ¦
 ¦  MOV [BX],0          ; OK!                                     ¦
 +-----------------------------------------------------------------+
```

Para ilustrar o uso da instrução MOV, eis um pedaço do código usado pela ROM-BIOS do IBM PS/2 Modelo 50Z para verificar a integridade dos registradores da CPU:

```
 +-----------------------------------------------------------------+
 ¦  ...                                                            ¦
 ¦  MOV AX,0FFFFh            ;Poe 0FFFFh em AX                     ¦
 ¦  MOV DS,AX                                                      ¦
 ¦  MOV BX,DS                                                      ¦
 ¦  MOV ES,BX                                                      ¦
 ¦  MOV CX,ES                                                      ¦
 ¦  MOV SS,CX                                                      ¦
 ¦  MOV DX,SS                                                      ¦
 ¦  MOV SI,DX                                                      ¦
 ¦  MOV DI,SI                                                      ¦
 ¦  MOV BP,DI                                                      ¦
 ¦  MOV SP,BP                                                      ¦
 ¦  ...                                                            ¦
 +-----------------------------------------------------------------+
```

Se o conteúdo de BP não for 0FFFFh então a CPU está com algum problema e o computador não pode funcionar! Os flags são testados de uma outra forma... :)

```
i------©
¦ XCHG ¦
È------¥
```

Esta instrução serve para trocarmos o conteúdo de um registrador pelo outro. Por exemplo:

```
 +-----------------------------------------------------------------+
 ¦  XCHG    AH,AL                                                  ¦
 +-----------------------------------------------------------------+
```

Se AH=1Ah e AL=6Dh, após esta instrução AH=6Dh e AL=1Ah por causa da troca...

Pode-se usar uma referência à memória assim como em MOV... com a mesma restrição de que um dos operandos TEM que ser um registrador. Não há possibilidade de usar um operando imediato.

MOVSB e MOUSW

Essas instruções suprem a deficiência de MOV quanto a movimentação de dados de uma posição de memória para outra diretamente. Antes de ser chamada os seguintes registradores tem que ser inicializados:

```
DS:SI   <- DS e SI têm o endereço fonte
ES:DI   <- ES e DI têm o endereço destino
```

Dai podemos executar MOVSB ou MOVSW.

MOVSB move um byte, enquanto MOVSW move um word (16 bits).

Os registradores SI e DI sao incrementados ou decrementados de acordo com o flag D (Direction) - Veja discussão sobre os flags na mensagem anterior. No caso de MOVSW, SI e DI serao incrementados (ou decrementados) de 2 posições de forma que DS:SI e ES:DI apontem sempre para a próxima word.

STOSB e STOSW

Essas instruções servem para armazenar um valor que está em AX ou AL (dependendo da instrução usada) no endereço apontado por ES:DI. Então, antes de ser chamada, os seguintes registradores devem ser inicializados:

```
AX      -> Valor a ser armazenado se usarmos STOSW
AL      -> Valor a ser armazenado se usarmos STOSB
ES:DI   -> Endereço onde o dado será armazenado
```

Depois da execução da instrução o registrador DI será incrementado ou decrementado de acordo com o flag D (Direction). DI será incrementado de 2 no case de usarmos STOSW, isto garante que ES:DI aponte para a proxima word.

LODSB e LODSW

Essas instruções servem para ler um valor que está no endereço apontado por DS:SI e armazená-lo em AX ou AL (dependendo da instrução usada). Então, antes de ser chamada, os seguintes registradores devem ser inicializados:

```
DS:SI   -> Endereço de onde o dado será lido
```

Depois da execução da instrução o registrador SI será incrementado ou decrementado de acordo com o flag D (Direction). No caso de usarmos LODSW, SI será incrementado de 2 para garantir que DS:SI aponte para a próxima word.

Outras instruções de manipulação de registros/dados

Existem ainda as instruções LEA, LES e LDS.

LEA:

LEA é, basicamente, igual a instrução MOV, com apenas uma diferença: o operando "fonte" é um endereço (precisamente: um "offset"). LEA simplesmente calcula o endereço e transfere para o operando "destino", de forma que as instruções abaixo sao equivalentes:

```
MOV     BX,100h
LEA     BX,[100h]
```

Porém, a instrução:

```
LEA     DX,[BX + SI + 10h]
```

Equivale a:

```
MOV     DX,BX
ADD     DX,SI         ; DX = DX + SI
ADD     DX,10h        ; DX = DX + 10h
```

Repare que apenas uma instrução faz o serviço de três!! Nos processadores 286 e 386 a diferença é significativa, pois, no exemplo acima, LEA gastará 3 (nos 286) ou 2 (nos 386) ciclos de máquina enquando o equivalente gastará 11 (nos 286) ou 6 (nos 386) ciclos de máquina! Nos processadores 8088/8086 a diferença não é tao grande...

Obs:

Consideremos cada ciclo de máquina seria, aproximadamente, num 386DX/40, algo em torno de 300ns - ou 0,0000003s. É uma medida empirica e não expressa a grandeza real (depende de uma série de fatores não considerados aqui!).

O operando "destino" é sempre um registrador. O operando "fonte" é sempre um endereço.

LDS e LES

Existe uma forma de carregar um par de registradores (segmento:offset) de uma só vez. Se quisermos carregar DS:DX basta usar a instrução LDS, caso o alvo seja ES, usa-se LES.

Suponhamos que numa posição da memória tenhamos um double word (númeroo de 32 bits) armazenado. A word mais significativa correspondendo a um segmento e a menos signigicativa a um offset (esse é o caso da tabela dos vetores de interrupção, que descreverei com poucos detalhes em uma outra oportunidade!). Se usamos:

```
LES BX,[SI]
```

O par ES:BX será carregado com o double word armazenado no endereço apontado por DS:SI (repare no segmento default que discutimos em um texto anterior!). A instrução acima é equivalente a:

```
MOV     BX,[SI+2]
MOV     ES,BX
MOV     BX,[SI]
```

De novo, uma instrução substitui três!

Manipulando blocos... parte I

As instruções MOVSB, MOVSW, STOSB, STOSW, LODSB e LODSW podem ser usadas para lidar com blocos de dados. Para isto, basta indicar no registrador CX a quantidade de dados a serem manipulados e acrescentar REP na frente da instruçao. Eis um trecho de uma pequena rotina que apaga o video em modo texto (80 x 25 colorido):

```
MOV AX,0B800h
MOD ES,AX           ; Poe em ES o segmento do vídeo
MOV DI,0            ; Começa no Offset 0
MOV AH,7            ; AH = atributo do caracter
                    ;      7 = cinza com fundo preto
MOV AL,' '          ; AL = caracter usado para apagar
MOV CX,2000         ; CX = contador (4000 bytes ou
                    ;      2000 words).
REP STOSW           ; Preenche os 2000 words com AX
```

O modificador REP diz a instrução que esta deve ser executada CX vezes. Note que a cada execução de STOSW o registrador DI apontará para a proxima word.

Suponha que queiramos mover 4000 bytes de alguma posição da memória para o video, preenchendo a tela com esses 4000 bytes:

```
MOV AX,0B800h
MOD ES,AX           ; Poe em ES o segmento do vídeo
MOV AX,SEG TABELA
MOV DS,AX           ; Poe em DS o segmento da tabela
MOV SI,OFFSET TABELA ; Começa no offset inicial da tabela
MOV DI,0            ; Começa no Offset 0
MOV CX,4000         ; CX = contador (4000 bytes)
REP MOVSB           ; Copia 4000 bytes de DS:SI para ES:DI
```

Nota: O modificador REP só pode ser preceder as seguintes instruções: MOVSB, MOVSW, STOSB, STOSW, LODSB, LODSW, CMPSB, CMPSW, SCASB, SCASW, OUTSB, OUTSW, INSB, INSW. As instruções nao vistas no texto acima serão detalhadas mais tarde...

Existem mais algumas instruções de manipulação de registradores/dados, bem como mais algumas de manipulação de blocos. Que ficarão para uma próxima mensagem.

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 05 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
i------------©
¦ ASSEMBLY V ¦
È------------¥
```

Depois de algumas instruções de movimentação de dados vou mostrar a mecânica da lógica booleana, bem como algumas instruções.

A lógica booleana baseia-se nas seguintes operações: AND, OR, NOT. Para simplificar a minha digitação vou usar a notação simplificada: & (AND), | (OR) e ~ (NOT). Essa notação é usada na linguagem C e em muitos manuais relacionados a hardware da IBM.

¦ Operação AND:

A operação AND funciona de acordo com a seguinte tabela-verdade:

```
          +-----------+
          ¦ S = A & B ¦
          ã---Đ---Đ---Á
          ¦ A ¦ B ¦ S ¦
          +---+---+---¦
          ¦ 0 ¦ 0 ¦ 0 ¦
          ¦ 0 ¦ 1 ¦ 0 ¦
          ¦ 1 ¦ 0 ¦ 0 ¦
          ¦ 1 ¦ 1 ¦ 1 ¦
          +-----------+
```

Note que o resultado (S) será 1 apenas se A "E" B forem 1.
Aplicando esta lógica bit a bit em operações envolvendo dois bytes obteremos um terceiro byte que será o primeiro AND o segundo:

```
+-------------------------------------------------------------------+
¦              A = 01010111b       B = 00001111b                  ¦
¦                                                                  ¦
¦              S = A & B ->    01010111b                          ¦
¦                            & 00001111b                           ¦
¦                              -------------                       ¦
¦                                 00000111b                        ¦
+-------------------------------------------------------------------+
```

Uma das utilidades de AND é resetar um determinado bit sem afetar os demais. Suponha que queira resetar o bit 3 de um determinado byte. Para tanto basta efetuar um AND do byte a ser trabalhado com o valor 11110111b (Apenas o bit 3 resetado).

Eis a sintaxe da instrução AND:

```
+-------------------------------------------------------------------+
¦     AND AL,11110111b                                             ¦
¦     AND BX,8000h                                                 ¦
¦     AND DL,CL                                                    ¦
¦     AND [DI],AH                                                  ¦
+-------------------------------------------------------------------+
```

Lembrando que o operando destino (o mais a esquerda) deve sempre ser um registrador ou uma referencia a memória. o operando a

direita (fonte) pode ser um registrador, uma referência a memória ou um valor imediato, com a restrição de que não podemos usar referências a memória nos dois operandos.

A instrução AND afeta os FLAGS Z, S e P e zera os flags Cy (Carry) e O (veja os flags em alguma mensagem anterior a esta).

¦ Operação OR:

S = A | B

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

Note que S será 1 se A "OU" B forem 1.
Da mesma forma que AND, aplicamos essa lógica bit a bit envolvendo um byte ou word através de uma instrução em assembly. Vejamos um exemplo da utilidade de OR:

```
A = 01010111b       B = 10000000b

S = A | B ->    01010111b
                | 10000000b
                -----------
                  11010111b
```

A operação OR é ideal para setarmos um determinado bit sem afetar os demais. No exemplo acima B tem apenas o bit 7 setado... depois da operação OR com A o resultado final foi A com o bit 7 setado! :)

A sintaxe de OR é a mesma que a de AND (obviamente trocando-se AND por OR). Os flags afetados são os mesmos da instrução AND!

¦ Operação NOT:

NOT simplesmente inverte todos os bits de um byte ou word:

S = ~A

A	S
0	1
1	0

A sintaxe da instrução em assembly é a seguinte:

```
NOT AL
NOT DX
NOT [SI]
```

¦ Operação XOR:

A operação XOR é derivada das três acima. A equação booleana que descreve XOR é:

```
+-------------------------------------------------------------------+
¦  S = (A AND ~B) OR (~A AND B) = A ^ B                              ¦
+-------------------------------------------------------------------+
```

Que na tabela-verdade fica:

```
                        +-----------+
                        ¦ S = A ^ B ¦
                        ã---Đ---Đ---Á
                        ¦ A ¦ B ¦ S ¦
                        +---+---+---¦
                        ¦ 0 ¦ 0 ¦ 0 ¦
                        ¦ 0 ¦ 1 ¦ 1 ¦
                        ¦ 1 ¦ 0 ¦ 1 ¦
                        ¦ 1 ¦ 1 ¦ 0 ¦
                        +-----------+
```

Uso o simbolo ^ para o XOR aqui. XOR funciona da mesma forma que OR, só que o resultado será 1 se APENAS A ou APENAS B for 1, melhor dizendo: Se ambos forem diferentes.

XOR é muito útil quando se quer inverter um determinado bit de um byte ou word sem afetar os outros:

```
+-------------------------------------------------------------------+
¦              A = 01010111b       B = 00001111b                  ¦
¦                                                                  ¦
¦              S = A ^ B ->    01010111b                           ¦
¦                            ^ 00001111b                           ¦
¦                              -------------                       ¦
¦                              01011000b                           ¦
+-------------------------------------------------------------------+
```

No exemplo acima invertemos apenas os quatro bits menos significativos de A.

A sintaxe e os flags afetados são os mesmos que AND e OR.

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 06 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
i-------------©
¦ ASSEMBLY VI ¦
È-------------¥
```

Instruções aritiméticas são o tópico de hoje. Já discuti, brevemente, os flags e os sistemas de numeração. Aqui vai uma aplicação prática:

¦ Soma:

A soma é feita através das instruções ADD e ADC. A diferença entre elas é que uma faz a soma normalmente e a outra faz a mesma coisa acrecentando o conteúdo do flag CARRY. Eis a sintaxe:

```
+-------------------------------------------------------------------+
¦  ADD AL,10h                                                       ¦
¦  ADC AH,22h                                                       ¦
¦                                                                   ¦
¦  ADD AX,2210h                                                     ¦
+-------------------------------------------------------------------+
```

As duas primeiras instruções fazem exatamente a mesma coisa que a terceira. Note que na primeiria somamos AL com 10h e o resultado ficará em AL (se ocorrer "vai um" nesta soma o flag CARRY será setado). A segunda instrução soma AH com 22h MAIS o carry resultante da primeira instrução e o resultado ficará em AH (novamente setando o flag carry se houver outro "vai um"!). A terceira instrução faz a mesma coisa porque soma 2210h a AX, ficando o resultado em AX e o possível "vai um" no carry.

Todos os flags são afetados após a execução de uma das instruções de soma, exceto: I, D e Trap.

¦ Subtração

Semelhante as instruções de soma, existem duas instruções de subtração: SUB e SBB. A primeira faz a subtração simples e a segunda faz a mesma coisa subtraindo também o conteúdo prévio do flag CARRY (como é uma subtração o CARRY é conhecido como BORROW!).

A sintaxe:

```
+-------------------------------------------------------------------+
¦  SUB AL,1                                                         ¦
¦  SBB AH,0                                                         ¦
¦                                                                   ¦
¦  SUB AX,1                                                        ¦
+-------------------------------------------------------------------+
```

Como no exemplo anterior, as duas primeiras instruções fazem exatamente o que a terceira faz... Os flags afetados seguem a mesma regra das instruções de soma!

¦ Incremento e decremento:

As instruções INC e DEC são usadas no lugar de ADD e SUB se quisermos incrementar ou decrementar o conteúdo de algum registrador (ou de uma posição de memória) de uma unidade. A sintaxe é simples:

```
DEC AX
INC BL
```

Os flags afetados seguem a mesma regra de uma instrução de soma ou uma de subtração!

Multiplicação:

Os processadores da família 80x86 possuem instruções de multiplicação e divisão inteiras (ponto flutuante fica pro 8087). Alguns cuidados devem ser tomados quando usarmos uma instrução de divisão (que será vista mais adiante!).

Uma coisa interessante com a multiplicação é que se multiplicarmos dois registradores de 16 bits obteremos o resultado necessariamente em 32 bits. O par de registradores DX e AX são usados para armazenar esse númeroo de 32 bits da seguinte forma: DX será a word mais significativa e AX a menos significativa.

Por exemplo, se multiplicarmos 0FFFFh por 0FFFFh obteremos: 0FFFE0001h (DX = 0FFFEh e AX = 0001h).

Eis a regra para descobrir o tamanho do restultado de uma operação de multiplicação:

A * B = M		
A	B	M
8 bits	8 bits	16 bits
16 bits	16 bits	32 bits

A multiplicação sempre ocorrerá entre o acumulador (AL ou AX) e um outro operando. Eis a sintaxe das instruções:

```
MUL BL      ; AX = AL * BL
IMUL CX     ; DX:AX = AX * CX
```

A primeira instrução (MUL) não considera o sinal dos operandos. Neste caso, como BL é de 8 bits, a multiplicação se dará entre BL e AL e o resultado será armazenado em AX.

A segunda instrução leva em consideração o sinal dos operandos e, como CX é de 16 bits, a multiplicação se dará entre CX e AX e o restultado será armazenado em DX e AX. Lembrando que o sinal de um númeroo inteiro depende do seu bit mais significativo!

Divisão:

Precisamos tomar cuidado com a divisão pelo seguinte motivo: Se o resultado não couber no registrador destino, um erro de "Division

by zero" ocorrerá (isto não está perfeitamente documentado nos diversos manuais que li enquanto estudava assembly 80x86... Vim a descobrir este 'macete' numa antiga edição da revista PC MAGAZINE americana). Outro cuidado é com o divisor... se for 0 o mesmo erro ocorrerá!

A divisão pode ser feita entre um númeroo de 32 bits e um de 16 ou entre um de 16 e um de 8, veja a tabela:

A / B = Q e resto		
A	B	Q e resto
32 bits	16 bits	16 bits
16 bits	8 bits	8 bits

Assim como na multiplicação o númeroo (dividendo) de 32 bits é armazenado em DX e AX.

Depois da divisão o quociente é armazenado em AL e o resto em AH (no caso de divisão 16/8 bits) ou o quociente fica em AX e o resto em DX (no caso de divisão 32/8 bits).

Exemplo da sintaxe:

```
DIV CX      ; AX=DX:AX / CX, DX=resto
IDIV BL     ; AL=AX / BL, AH=resto
```

O primeiro caso é uma divisão sem sinal e o segundo com sinal. Note os divisores (CX e BL no nosso exemplo).

Na divisão 16/8 bits o dividendo é armazenado em AX antes da divisão... No caso de 32/8 bits DX e AX são usados...

Mais um detalhe: Os flags, depois de uma multiplicação ou divisão não devem ser considerados.

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 07 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
i--------------©
¦ ASSEMBLY VII ¦
È--------------¥
```

Algumas instruções afetam somente aos flags. Dentre elas, as mais utilizadas são as instruções de comparação entre dois dados.

¦ Comparações aritiméticas:

A instrução CMP é usada quando se quer comparar dois dados, afetando somente aos flags. Eis a sintaxe:

```
+-------------------------------------------------------------------+
¦  CMP AL,1Fh                                                       ¦
¦  CMP ES:[DI],1                                                    ¦
¦  CMP AX,[SI]                                                      ¦
+-------------------------------------------------------------------+
```

Esta instrução faz a subtração entre o operando mais a esquerda e o mais a direita, afetando somente os flags. Por exemplo, se os dois operandos tiverem valores iguais a subtração dará valores iguais e o flag de ZERO será 1. Eis a mecânica de CMP:

```
+-------------------------------------------------------------------+
¦  CMP AL,1Fh   ; AL - 1Fh, afetando somente os Flags                ¦
+-------------------------------------------------------------------+
```

¦ Comparações lógicas:

A instrução TEST é usada quando se quer comparar o estado de determinados bits de um operando. Eis a sintaxe:

```
+-------------------------------------------------------------------+
¦  TEST AL,10000000b                                                ¦
¦  TEST [BX],00001000b                                              ¦
+-------------------------------------------------------------------+
```

Esta instrução faz um AND com os dois operados, afetando apenas os flags. Os flags Z, S e P são afetados, os flags O e C serão zerados.

¦ Instruções que mudam o estado dos flags diretamente:

CLC - Abreviação de CLear Carry (Zera o flag Carry).
CLD - Abreviação de CLear Direction (Ajusta flag de direção em zero, especificando o sentido correto para instruções de bloco).
CLI - Abreviação de CLear Interrupt (Mascara flag de interrupção, não permitindo que a CPU reconheça as interrupções mascaráveis).
CMC - Abreviação de CoMplement Carry (Inverte o flag de carry).
STC - Abreviação de SeT Carry (Faz carry = 1).
STD - Abreviação de SeT Direction (flag de direção setado - indica que as instruções de bloco incrementarão os seus pointers no sentido contrário - de cima para baixo).
STI - Abreviação de SeT Interrupt (Faz com que a CPU volte a reconhecer as interrupções mascaráveis).

Uma interrupção é um "desvio" feito pela CPU quando um dispositivo requer a atenção da mesma. Por exemplo, quando você digita uma tecla, o circuito do teclado requisita a atenção da CPU, que por sua vez, para o que está fazendo e executa uma rotina correspondente à requisição feita pelo dispositivo (ou seja, a rotina da interrupção). Ao final da rotina, a CPU retorna à tarefa que estava desempenhando antes da interrupção. Nos PCs, TODAS as interrupções são mascaráveis (podem ser ativadas e desativadas quando quisermos), com a única excessão da interrupçao de checagem do sistema (o famoso MEMORY PARITY ERROR é um exeplo!).

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 08 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
i---------------©
¦ ASSEMBLY VIII ¦
È---------------¥
```

Veremos agora as instruções de controle de fluxo de programa.

A CPU sempre executa instruções em sequência, a não ser que encontre instruções que "saltem" para outra posição na memória.

Existem diversas formas de "saltar" para um determinado endereço:

¦ Salto incondicional:

A instrução JMP simplesmente salta para onde se quer. Antes de apresentar a sintaxe, um detalhe sobre codificaçao: O operando da instrução JMP é um endereço na memória, mas, como usaremos sempre um compilador assembly, necessitamos criar um "rotulo" ou "label" para onde o salto será efetuado... O compilador trata de calcular o endereço pra gente.

Eis a sintaxe de JMP:

```
+-------------------------------------------------------------------+
¦      JMP Aqui2                                                    ¦
¦ Aqui1:                                                           ¦
¦      JMP Aqui3                                                    ¦
¦ Aqui2:                                                           ¦
¦      JMP Aqui1                                                    ¦
¦ Aqui3:                                                           ¦
+-------------------------------------------------------------------+
```

Os "labels" são sempre seguidos de dois-pontos. Note, no pedaço de código acima, a quebra da sequência de execução.

¦ Salto incondicional:

Diferente de JMP, temos instruções que realizam um salto somente se uma condição for satisfeita. Para isso, usa-se os flags. A sintaxe dessas instruções depende da condição do flag que se quer testar. Eis a listagem dessas instruções:

- JZ "label" -> Salta se flag Z=1
- JNZ "label" -> Salta se flag Z=0
- JC "label" -> Salta se flag C=1
- JNC "label" -> Salta se flag C=0
- JO "label" -> Salta se flag O=1
- JNO "label" -> Salta se flag O=0
- JPO "label" -> Salta se flag P=0 (paridade impar)
- JPE "label" -> Salta se flag P=1 (paridade par)
- JS "label" -> Salta se flag S=1
- JNS "label" -> Salta se flag S=0

Existem ainda mais saltos condicionais para facilitar a vida do programador:

- JE "label" -> Jump if Equal (mesmo que JZ)
- JNE "label" -> Jump if Not Equal (mesmo que JNZ)
- JA "label" -> Jump if Above (salta se acima)
- JB "label" -> Jump if Below (salta se abaixo)
- JAE "label" -> Jump if Above or Equal (salta se acima ou =)
- JBE "label" -> Jump if Below of Equal (salta se abaixo ou =)
- JG "label" -> Jump if Greater than (salta se >)
- JL "label" -> Jump if Less than (salta se <)
- JGE "label" -> Jump if Greater than or Equal (salta se >=)
- JLE "label" -> Jump if Less or Equal (salta se <=)

A diferença entre JG e JA, JL e JB é:

- JA e JB são relativos a comparações sem sinal.
- JG e JL são relativos a comparações com sinal.

Os saltos condicionais têm uma desvantagem com relação aos saltos incondicionais: O deslocamento é relativo a posição corrente, isto é, embora no nosso código o salto se dê na posição do "label" o assembler traduz esse salto para uma posição "x" bytes para frente ou para tras em relação a posição da instrução de salto... e esse númeroo "x" está na faixa de -128 a 127 (traduzindo isso tudo pra quem não entendeu: Não é possível saltos muito longos com instruções de salto condicionais... salvo em casos especiais que explicarei mais tarde!).

Existe ainda a instrução JCXZ. Essa instrução salta se o registrador CX for 0.

Mais uma instrução: LOOP

A instrução LOOP salta para um determinado endereço se o registrador CX for diferente de zero e, antes de saltar, decrementa CX. Um exemplo do uso desta instrução:

```
     SUB AL,AL       ;AL = 0
     SUB DI,DI       ;DI = 0
     MOV CX,1000     ;CX = 1000
Loop1:
     MOV BYTE PTR ES:[DI],0  ;Poe 0 em ES:DI
     INC DI          ;Incrementa o offset (DI)
     LOOP Loop1      ;Repete ate' que CX seja 0
```

Essa rotina preenche os 1000 bytes a partir de ES:0 com 0. O modificador "BYTE PTR" na frente de ES:[DI] resolve uma ambiguidade: Como podemos saber se a instrução "MOV ES:[DI],0" escreverá um byte ou um word? Por default, o compilador assume word, por isso temos que usar o modificador indicando que queremos byte.

Repare que o pedaço entre "Loop1" e o final da rotina equivale a uma instrução "REP STOSB".

Podemos também especificar uma instrução LOOP condicional, basta acrescentar 'Z' ou 'NZ' (ou os equivalentes 'E' ou 'NE') no fim. Isto quer dizer: Salte ENQUANTO CX for ZERO (Z) ou NãO for ZERO (NZ). A instrução LOOP sem condição é a mesma coisa que LOOPNZ ou LOOPNE!

¦ Chamadas a sub-rotinas:

A instrução CALL funciona como se fosse a instrução GOSUB do

velho BASIC. Ela salta para a posição especificada e quando a instrução RET for encontrada na sub-rotina a CPU salta de volta para a próxima instrução que segue o CALL. A sintaxe:

```
CALL "label"
```

Eis um exemplo:

```
     MOV AL,9        ;Poe numero em AL
     CALL ShowNumber ;Salta para a subrotina
     ...

ShowNumber:
     ...
     RET     ;Retorna
```

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 09 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
i-------------©
¦ ASSEMBLY IX ¦
È-------------¥
```

O assunto de hoje é INTERRUPÇõES. Como já disse antes, uma interrupção é uma requisição da atenção da CPU por um dispositivo (por exemplo o teclado, quando apertamos uma tecla!). A CPU INTERROMPE o processamento normal e salta para a rotina que "serve" a interrupção requisitada, retornando ao ponto em que estava ANTES da interrupção quando finalizar a rotina de interrupção. Assim funciona a nível de hardware.

A novidade nos processadores INTEL da série 80x86 é que existem instruções assembly que EMULAM a requisição de uma interrução. Essas instruções nada mais são que um "CALL", mas ao invés de usarmos um endereço para uma subrotina, informamos o índice (ou o código) da interrupção requisitada e a CPU se comportará como se um dispositivo tivesse requisitado a interrupção...

As rotinas do DOS e da BIOS são chamadas por essas instruções. Na realidade, este artificio da família INTEL facilita muito o trabalho dos programadores porque não precisamos saber onde se encontram as rotinas da BIOS e do DOS na memória... Precisamos saber apenas o índice da interrupção de cada uma das rotinas... o endereço a CPU calcula para nós!

Eis a sintaxe da instrução:

```
+-------------------------------------------------------------------+
¦  INT 21h                                                          ¦
¦  INT 10h                                                          ¦
+-------------------------------------------------------------------+
```

Onde 21h e 10h são índices.

A CPU sabe para onde saltar porque no inicio da memória de todo PC tem uma tabela conhecida como "Tabela dos vetores de interrupção". A CPU, de posse do índice na instrução INT, "pega" o endereço correspondente a esse índice nessa tabela e efetua um CALL diferente (porque o fim de uma rotina de interrupção tem que terminar em IRET e não em RET - IRET é o RET da rotina de interrupção - Interrupt RETurn).

Por exemplo... Se precisamos abrir um arquivo, o trabalho é enviado ao DOS pela interrupçao de indice 21h. Se queremos ler um setor do disco, usamos a interrupçao de indice 13h, etc... Mas, nao use a instruçao INT sem saber exatamente o que está fazendo, ok? Pode ter resultados desastrosos!

Uma descrição da maioria das interrupções de software disponíveis nos PCs compatíveis está disponível no livro "Guia do programador para PC e PS/2" de Peter Norton (recomendo a aquisição deste livro! De preferencia a versao americana!). Ou, se preferir "literatura eletronica" recomendo o arquivo HELPPC21.ZIP (v2.1), disponivel em qualquer bom BBS... Ainda assim pedirei para o RC do ES (RBT) para disponibiliza-lo para FREQ aos Sysops interessados em adquiri-lo.

Quanto as interruções de hardware (as famosas IRQs!)... é assunto meio complexo no momento e requer um bom conhecimento de eletronica digital e do funcionamento do micrprocessador... no futuro (próximo, espero!) abordarei esse assunto.

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 10 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

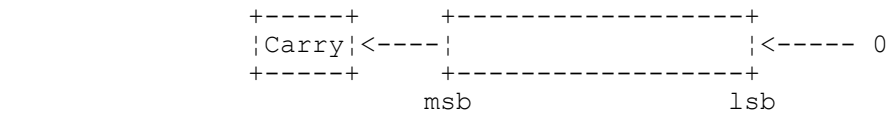
```
i------------©
¦ ASSEMBLY X ¦
È------------¥
```

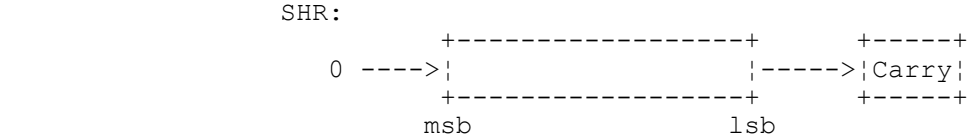
Mais instruções lógicas... Falta-nos ver as intruções de deslocamento de bits: SHL, SHR, SAL, SAR, ROL, ROR, RCL e RCR.

A última letra nas instruções acima especifica o sentido de rotação (R = Right -> direita, L = Left -> esquerda).

Para exemplificar a mecânica do funcionamento dessas instruções recorrerei a graficos (fica mais fácil assim).

```
Í-----------À
¦ SHL e SHR ¦
Ë-----------¢
```



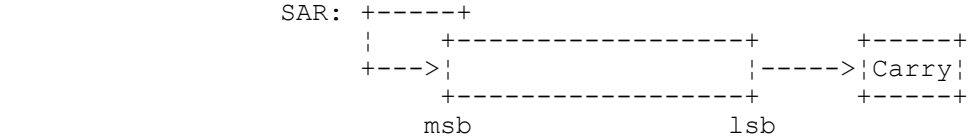


SHR e SHL fazem o deslocamento dos bits em direção ao flag Carry e acrescentam 0 no lugar do último bit que foi deslocado. Essa operação tem o mesmo efeito de multiplicar por 2 (SHL) ou dividir por 2 (SHR) um valor. Com a vantagem de não gastar tanto tempo quanto as instruções DIV e MUL.

SHR é a abreviação de SHift Right, enquando SHL é a de SHift Left.

```
Í-----------À
¦ SAL e SAR ¦
Ë-----------¢
```

SAL funciona da mesma maneira que SHL.



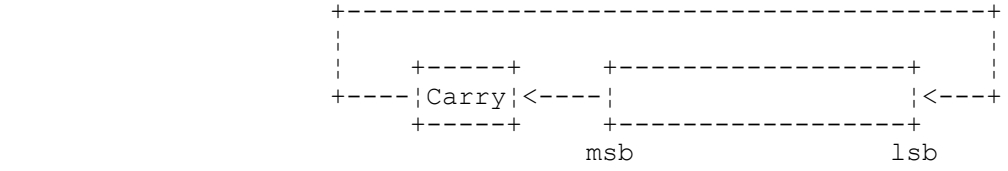
SAR desloca todos os bits para a direita (o lsb vai para o flag carry) e repete o conteúdo do antigo último bit (que foi deslocado).

SAR é a abreviação de SHift Arithmetic Right. Sendo um deslocamento aritimético, não poderia de desconsiderar o sinal do

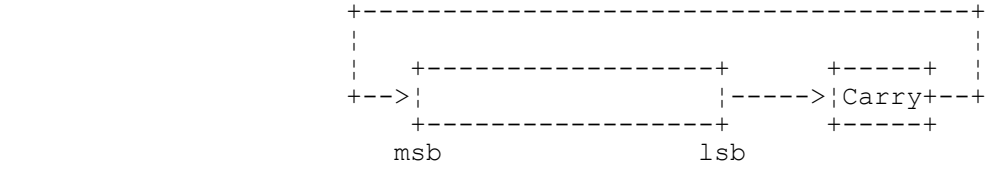
dado deslocado (dai o motivo de repetir o bit mais significativo!).

RCL e RCR

RCL:



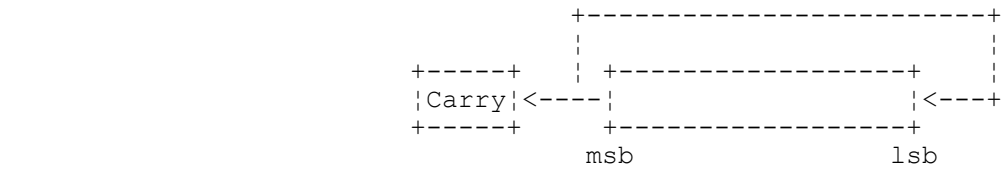
RCR:



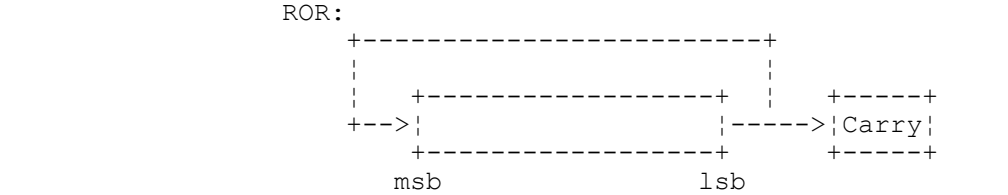
RCR e RCL rotacionam o dado "passando pelo carry". Isto significa que o bit menos significativo (no caso de ROR) será colocado no flag de carry e que o conteúdo antigo deste flag será colocado no bit mais significativo do dado.

ROL e ROR

ROL:



ROR:



Aqui a rotaçao e' feita da maneira correta... o flag de carry apenas indica o ultimo bit que "saiu" e foi para o outro lado...

A sintaxe dessas instruções é a seguinte:

```
SHL AX,1
SHR BL,1
RCL DX,CL
ROL ES:[DI],CL
```

Note que o segundo operando é um contador do número de rotações

ou shifts serão efetuadas. Nos microprocessadores 80286 em diante pode-se usar um valor diferente de 1, no 8088/8086 não pode!

Repare também que podemos usar APENAS o registrador CL como operando da direita se quisermos usar algum registrador!

```
              i-----------------------------------------------©
              ¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 11 ¦
              È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
i-------------©
¦ ASSEMBLY XI ¦
È-------------¥
```

Mais instruções de comparação...

¦ CMPSB e CMPSW

Essas instruções comparam (da mesma forma que CMP) o conteúdo da memória apontada por DS:SI com o conteúdo apontado por ES:DI, afetando os flags. Com isso, soluciona-se a limitação da instrução CMP com relação aos dois operandos como referências à memória!

Lembre-se que DS:SI é o operando implicito FONTE, enquanto ES:DI é o destino. A comparação é feita de ES:DI para DS:SI. A rotina abaixo é equivalente a CMPSB:

```
 +-------------------------------------------------------------------+
 ¦  MOV AL,ES:[DI]                                                   ¦
 ¦  CMP AL,[SI]                                                      ¦
 ¦  INC SI                                                           ¦
 ¦  INC DI                                                           ¦
 +-------------------------------------------------------------------+
```

Existe um pequenino erro de lógica na rotina acima, mas serve aos nossos propósitos de ilustrar o que ocorre em CMPSB.

SI e DI serão incrementados (ou decrementados, depende do flag de direção) depois da operação, e o incremento (ou decremento) dependerá da instrução... Lembre-se que CMPSB compara Bytes e CMPSW compara Words.

¦ SCASB e SCASW

Essas instruções servem para comparar (da mesma forma que CMP o faz) o conteúdo da memória apontado por DS:SI com o registrador AL (no caso de SCASB) ou AX (no caso de SCASW). Os flags são afetados e SI é incrementado (ou decrementado) de acordo com a instrução usada.

```
+-------------------------------------------------------------------+
¦Comparando blocos de memória:                                      ¦
+-------------------------------------------------------------------+
```

Podemos usar CMPS? e SCAS? (onde ? e' B ou W) em conjunto com REP para compararmos blocos (CMPS?) ou procurar por um determinado dado num bloco (SCAS?). A diferença aqui é que podemos fornecer uma condição de comparação ou busca.

Acrescentando o modigicador REP, precisamos dizer à uma dessas instruções a quantidades de dados que queremos manipular... fazemos isso através do registrador CX (assim como fizemos com LODS? e STOS?):

```
;Certifica-se do sentido crescente!
CLD

;Obtém o segmento da linha de comando e coloca em DS
MOV   AX,SEG LINHA_DE_COMANDO
MOV   DS,AX

;Obtém o offset inicial da linha de comando
MOV   SI,OFFSET LINHA_DE_COMANDO

;Procura, no máximo por 128 bytes
MOV   CX,128

;Procuraremos por um espaço.
MOV   AL,' '

REPNE SCASB
```

Esse fragmento de código ilustra o uso de SCASB com blocos. O modificador REPNE significa (REPete while Not Equal - Repete enquanto não for igual). REPNE garante que o byte vai ser procurado por toda a linha de comando até que o primeiro espaço seja encontrado. Se não houver espaços na linha, então, depois de 128 bytes de procura, o registrador CX estará zerado (já que é decrementado a cada byte comparado).

Esta é outra característica das instruções que manipulam blocos (as que são precedidas de REP, REPNE ou REPE): O contador é decrementado a cada operação da instrução associada (no nosso caso SCASB), bem como os demais operandos implicitos (SI no caso acima) é incrementado a cada passo.

Se quisermos encontrar o primeiro byte DIFERENTE de espaço na rotina acima, basta trocar REPNE por REPE (Repete enquanto for IGUAL).

REPE e REPNE não foram mencionados antes porque não funcionam com LODS? e STOS?.

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 12 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

ASSEMBLY XII

A partir de agora veremos, resumidamente, como desenvolver funções/procedures em assembly no mesmo código PASCAL.

O TURBO PASCAL (a partir da versao 6.0) fornece algumas palavras-chave dedicadas à construção de rotinas assembly in-line (esse recurso é chamado de BASM nos manuais do TURBO PASCAL - BASM é a abreviação de Borland ASseMbler).

Antes de começarmos a ver o nosso primeiro código em assembly vale a pena ressaltar alguns cuidados em relação a codificação de rotinas assembly em TURBO PASCAL... As nossas rotinas devem:

¦ Preservar sempre o conteúdo dos registradores DS, BP e SP.
¦ Nunca modificar, diretamente, o conteúdo dos registradores CS, IP e SS.

O motivo dessas restrições é que os registradores BP, SP e SS são usados na obtenção dos valores passados como parametros à função/procedure e na localização das variaveis globais na memória. O registrador DS é usado por todo o código PASCAL e aponta sempre para o segmento de dados corrente (o qual não sabemos onde se encontra... deixe que o código PASCAL tome conta disso!).

Com relação ao conteúdo de CS e IP, não é uma boa prática (nem mesmo em códigos assembly puros) alterar o seus valores. Deixe que as instruções de salto e chamada de subrotinas façam isso por você!).

Os demais registradores podem ser alterados a vontade.

A função HexByte() abaixo é um exemplo de função totalmente escrita em assembly... Ela toma um valor de 8 bits e devolve uma string de 2 bytes contendo o valor hexadecimal desse parametro:

```
¦  FUNCTION    HexByte(Data : Byte) : String; ASSEMBLER;          ¦
¦  ASM                                                           ¦
¦      LES     DI,@Result   { Aponta para o inicio da string. }  ¦
¦                                                               ¦
¦      MOV     AL,2         { Ajusta tamanho da string em 2.  }  ¦
¦      STOSB                                                    ¦
¦                                                               ¦
¦      MOV     AL,Data      { Pega o dado a ser convertido.   }  ¦
¦                                                               ¦
¦      MOV     BL,AL        { Salva-o em BL.                  }  ¦
¦      SHR     AL,1         { Para manter compatibilidade com }  ¦
¦      SHR     AL,1         { os microprocessadores 8088/8086 }  ¦
¦      SHR     AL,1         { nao é prudente usar SHR AL,4.   }  ¦
¦      SHR     AL,1                                             ¦
¦      ADD     AL,'0'       { Soma com ASCII '0'.             }  ¦
¦      CMP     AL,'9'       { Maior que ASCII '9'?            }  ¦
¦      JBE     @NoAdd_1     { ... Nao é, então nao soma 7.    }  ¦
```

```
    ADD     AL,7          { ... É, então soma 7.            }
  @NoAdd_1:
    MOV     AH,AL         { Salva AL em AH.                 }

    MOV     AL,BL         { Pega o valor antigo de AL em BL.}
    AND     AL,1111B      { Zera os 4 bits superiores de AL.}
    ADD     AL,'0'        { Soma com ASCII '0'.             }
    CMP     AL,'9'        { Maior que ASCII '9'?            }
    JBE     @NoAdd_2      { ... Nao é, então nao soma 7.    }
    ADD     AL,7          { ... É, então soma 7.            }
  @NoAdd_2:

    XCHG    AH,AL         { Trocar AH com AL para gravar na }
    STOSW                 { ordem correta.                  }
  END;
```

A primeira linha é a declaração da função seguida da diretiva ASSEMBLER (informando que a função TODA foi escrita em assembly!). A seguir a palavra-chave ASM indica o inicio do bloco assembly até que END; marque o fim da função...

A primeira linha do código assembly é:

```
    LES     DI,@Result
```

Quando retornamos uma string numa função precisamos conhecer o endereço do inicio dessa string. A variável @Result contém um pointer que aponta para o inicio da string que será devolvida numa função. Esse endereço é sempre um endereço FAR (ou seja, no formato SEGMENTO:OFFSET).

A seguir inicializamos o tamanho da string em 2 caracteres:

```
    MOV     AL,2
    STOSB
```

Note que STOSB vai gravar o conteúdo de AL no endereço apontado por ES:DI, ou seja, o endereço apontado por @Result, e logo após DI é incrementado, apontando para a primeira posição valida da string.

O método que usei para gerar uma string hexadecimal é o seguinte:

- Pegamos o parametro 'Data' e colocamos em AL.
- Salva-se o conteúdo de AL em BL para que possamos obter os 4 bits menos significativos sem termos que ler 'Data' novamente!
- Com AL fazemos:
  - Desloca-se AL 4 posiçoes para a direita, colocando os 4 bits mais significativos nos 4 menos significativos e preenchendo os 4 mais significativos com 0B.
  - (a)- Soma-se o valor do ASCII '0' a AL.
  - (b)- Verifica-se se o resultado é maior que o ASCII '9'.
    - Se for, somamos 7.
  - Salvamos o conteúdo de AL em AH.
- Recuperamos o valor antigo de AL que estava em BL.
- Com AL fazemos:
  - Zeramos os 4 bits mais significativos para obtermos apenas os 4 menos significativos em AL.
  - Repetimos (a) e (b)

- Trocamos AL com AH e gravamos AX com STOSB

A primeira pergunta é: Porque somar 7 quando o resultado da soma com o ASCII '0' for maior que o ASCII '9'? A resposta pode ser vista no pedaço da tabela ASCII abaixo:

```
+-------------------------------------------------------------+
¦         0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F       ¦
¦                           +-----------+                     ¦
¦                         E esses 7 bytes ?                   ¦
+-------------------------------------------------------------+
```

Observe que depois do ASCII '9' segue o ASCII ':' ao invés do ASCII 'A', como é desejado... Entao, se o resultado da soma dos 4 bits menos signficativos (que varia de 0000B até 1111B - ou de 0 a 15) com o ASCII '0' for maior que o ASCII '9' precisamos compensar a existencia dos 7 caracteres indesejáveis!

Imagine que AL seja 0. Somando o ASCII '0' (que equivale ao númeroo 30h) a AL obteriamos:

```
+-------------------------------------------------------------+
¦  AL = 0010B = 2h                                            ¦
¦  AL = 2h + '0'                                              ¦
¦  AL = 2h + 30h                                              ¦
¦  AL = 32h = '2'                                             ¦
+-------------------------------------------------------------+
```

Imagine agora que AL seja 1011B. Fazendo as mesmas contas obteriamos AL = 3Bh (que é a mesma coisa que o ASCII ';'. No entando, 3Bh é maior que o ASCII '9' (ou seja, 39h)... Então:

```
+-------------------------------------------------------------+
¦  AL = ';' = 3Bh                                             ¦
¦  AL = 3Bh + 7h                                              ¦
¦  AL = 42h = 'B'                                             ¦
+-------------------------------------------------------------+
```

A outra coisa que você poderia me perguntar é o porque eu usei a instrução XCHG AH,AL no final do código. A resposta é simples... Os microprocessadores da INTEL gravam words na memória da seguinte maneira:

```
+-------------------------------------------------------------+
¦  Word = FAFBh                                               ¦
¦  Na memória: FBh FAh                                        ¦
+-------------------------------------------------------------+
```

Não importa se o seu computador seja um Pentium ou um XT... A memória é sempre dividida em BYTES. A CPU apenas "le" um conjunto maior de bytes de acordo com a quantidade de bits da sua CPU. Por exemplo, os microprocessadores 8086 e 80286 são CPUs de 16 bits e por isso conseguem ler 2 bytes (8 bits + 8 bits = 16 bits) de uma só vez... As CPUs 386 e 486 são de 32 bits e podem ler de uma só vez 4 bytes!

Esse conjunto de bytes que a CPU pode enxergar é sempre armazenado da forma contrária do que os olhos humanos leem... O byte menos significativo SEMPRE vem ANTES do mais significativo. No caso de um DOUBLEWORD (ou numero de 32 bits de tamanho) o formato é o mesmo... Exemplo:

```
+------------------------------------------------------------------+
¦  númeroo = FAFBFCFDFEh                                           ¦
¦  Na memória: FE FD FB FA                                         ¦
+------------------------------------------------------------------+
```

Analizando a rotina HexByte() a gente ve que AH tem o byte mais significativo e AL o menos significativo. Como o menos significativo vem sempre antes do mais significativo fiz a troca de AH com AL para que o númeroo HEXA seja armazenado de forma correta na memória (string). Um exemplo: Suponha que o você passe o valor 236 à função HexByte():

```
+------------------------------------------------------------------+
¦  Valor = 236 ou ECh                                              ¦
¦  Até antes de XCHG AH,AL:    AH = ASCII 'E'                      ¦
¦                              AL = ASCII 'C'                      ¦
+------------------------------------------------------------------+
```

Se não tivessemos a instrução XCHG AH,AL e simplesmente usassemos o STOSW (como está no código!) AH seria precedido de AL na memória (ou na string!), ficariamos com uma string 'CE'! Não me lembro se já falei que o L de AL significa LOW (ou menos significativo!) e H de AH significa HIGH (ou mais significativo), portanto AL e AH são, respectivamente, os bytes menos e mais significativos de AX!

Não se importe em coloca um RET ao fim da função, o TURBO PASCAL coloca isso sozinho...

Você deve estar se perguntando porque não fiz a rotina de forma tal que a troca de AH por AL não fosse necessária... Well... Fiz isso pra ilustrar a forma como os dados são gravados na memória! Retire XCHG AH,AL do código e veja o que acontece! Um outro bom exercício é tentar otimizar a rotina para que a troca não seja necessária...

E... para fechar a rotina, podemos aproveitar HexByte() para construir HexWord():

```
+------------------------------------------------------------------+
¦  Function HexWord(Data : Word) : String;                         ¦
¦  Var H, L : String;                                              ¦
¦  Begin                                                           ¦
¦      H := HexByte(HIGH(Data));                                   ¦
¦      L := HexByte(LOW(Data));                                    ¦
¦      HexWord := H + L;                                           ¦
¦  End;                                                            ¦
+------------------------------------------------------------------+
```

HexDoubleWord() eu deixo por sua conta :)

Aguardo as suas duvidas...

```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula N° 13 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
+---------------+
¦ ASSEMBLY XIII ¦
+---------------+
```

Algumas pessoas, depois de verem o código-exemplo do texto anterior, desenvolvido para ser compilado em TURBO PASCAL, me perguntaram: "E quanto ao C?!". Well... aqui vão algumas técnicas para codificação mista em C...

Antes de começarmos a dar uma olhada nas técnicas, quero avisar que meu compilador preferido é o BORLAND C++ 3.1. Ele tem algumas características que não estão presentes do MicroSoft C++ 7.0 ou no MS Visual C++! Por exemplo, O MSC++ ou o MS-Visual C++ não tem "pseudo"-registradores (que ajudam um bocado na mixagem de código, evitando os "avisos" do compilador).

Mesmo com algumas diferenças, você poderá usar as técnicas aqui descritas... As regras podem ser usadas para qualquer compilador que não gere aplicações em modo protegido para o MS-DOS.

¦ Regras para a boa codigificação assembly em C

Assim como no TURBO PASCAL, devemos:

* Nunca alterar CS, DS, SS, SP, BP e IP.
* Podemos alterar com muito cuidado ES, SI e DI
* Podemos alterar sempre que quisermos AX, BX, CX, DX

O registrador DS sempre aponta para o segmento de dados do programa... Se a sua função assembly acessa alguma variável global, e você tiver alterado DS, a variável que você pretendia acessar não estará disponível! Os registradores SS, SP e BP são usados pela linguagem para empilhar e desempilhar os parametros e variáveis locais da função na pilha... altera-los pode causar problemas! O par de registradores CS:IP nao deve ser alterado porque indica a próxima posição da memória que contém uma instrução assembly que será executada... Em qualquer programa "normal" esses últimos dois registradores são deixados em paz.

No caso dos registradores ES, SI e DI, o compilador os usa na manipulação de pointers e quando precisa manter uma variável num registrador (quando se usa a palavra-reservada "register" na declaração de uma variável, por exemplo!). Dentro de uma função escrita puramente em assembly, SI e DI podem ser alterados a vontade porque o compilador trata de salva-las na pilha (via PUSH SI e PUSH DI) e, ao término da função, as restaura (via POP DI e POP SI). A melhor forma de se saber se podemos ou não usar um desses registradores em um código mixto é compilando o programa e gerando uma listagem assembly (no BORLAND C++ isso é feito usando-se a chave -S na linha de comando!)... faça a análize da função e veja se o uso desses registradores vai prejudicar alguma outra parte do código!

Se você não quer ter essa dor de cabeça, simplesmente salve-os antes de usar e restaure-os depois que os usou!

¦ Modelamento de memória:

O mais chato dos compiladores C/C++ para o MS-DOS é o modelamento de memória, coisa que não existe no TURBO PASCAL! Digo "chato" porque esse recurso, QUE É MUITO UTIL, nos dá algumas dores de cabeça de vez em quando...

Os modelos COMPACT, LARGE e HUGE usam, por default, pointers do tipo FAR (segmento:offset). Os modelos TINY, SMALL e MEDIUM usam, por default, pointers do tipo NEAR (apenas offset, o segmento de dados é assumido!).

A "chatisse" está em criarmos códigos que compilem bem em qualquer modelo de memória. Felizmente isso é possível graças ao pre-processador:

```
#if defined(__TYNY__) || defined(__SMALL__) || defined(__MEDIUM__)
/* Processamento de pointers NEAR */
#else
/* Processamento dos mesmos pointers... mas, FAR! */
#endif
```

Concorda comigo que é meio chato ficar enchendo a listagem de diretivas do pré-processador?... C'est la vie!

¦ C + ASM

Os compiladores da BORLAND possuem a palavra reservada "asm". Ela diz ao compilador que o que a segue deve ser interpretado como uma instrução assembly. Os compiladores da MicroSoft possuem o "_asm" ou o "__asm". A BORLAND ainda tem uma diretiva para o pré-processador que é usada para indicar ao compilador que o código deve ser montado pelo TURBO ASSEMBLER ao invés do compilador C/C++:

```
¦ #pragma inline
```

Você pode usar isto ou então a chave -B da linha de comando do BCC... funciona da mesma forma! Você deve estar se perguntando porque usar o TURBO ASSEMBLER se o próprio compilador C/C++ pode compilar o código... Ahhhhh, por motivos de COMPATIBILIDADE! Se você pretende que o seu código seja compilável no TURBO C 2.0, por exemplo, deve incluir a diretiva acima!! Além do mais, o TASM faz uma checagem mais detalhada do código assembly do que o BCC...

Eis um exemplo de uma funçãozinha escrita em assembly:

```
¦ int f(int X)
¦ {
¦     asm mov     ax,X    /* AX = parametro X */
¦     asm add     ax,ax   /* AX = 2 * AX */
¦     return _AX;         /* retorna AX */
¦ }
```

Aqui segue mais uma regra:

¦ Se a sua função pretende devolver um valor do tipo "char" ou

"unsigned char", coloque o valor no registrador AL e (nos compiladores da BORLAND) use "return _AL;"

¦ Se a sua função pretende devolver um valor do tipo "int" ou "unsigned int", coloque o valor no registrador AX e (também nos compiladores da BORLAND) use "return _AX;"

A última linha da função acima ("return _AX;") não é necessária, mas se não a colocarmos teremos um aviso do compilador, indicando que "a função precisa retornar um 'int'". Se você omitir a última linha (é o caso dos compiladores da MicroSoft que não tem pseudo-registradores) e não ligar pros avisos, a coisa funciona do mesmo jeito.

Agora você deve estar querendo saber como devolver os tipos "long", "double", "float", etc... O tipo "long" (bem como "unsigned long") é simples:

¦ Se a sua função pretende devolver um valor do tipo "long" ou "unsigned long", coloque os 16 bits mais significativos em DX e os 16 menos significativos em AX.

Não existe uma forma de devolvermos DX e AX ao mesmo tempo usando os pseudo-registradores da Borland, então prepare-se para um "aviso" do compilador...

Os demais tipos não são inteiros... são de ponto-flutuante, portanto, deixe que o compilador tome conta deles.

¦ Trabalhando com pointers e vetores:

Dê uma olhada na listagem abaixo:

```
+--------------------------------------------------------------------+
¦   unsigned ArraySize(char *str)                                     ¦
¦   {                                                                 ¦
¦#if defined(__TYNY__) || defined(__SMALL__) || defined(__MEDIUM__)   ¦
¦       asm mov     si,str  /* STR = OFFSET apenas */                ¦
¦#else                                                               ¦
¦       asm push    ds                                               ¦
¦       asm lds     si,str  /* STR = SEGMENTO:OFFSET */              ¦
¦#endif                                                              ¦
¦                                                                    ¦
¦       asm mov     cx,-1                                            ¦
¦ContinuaProcurando:                                                ¦
¦       asm inc     cx                                               ¦
¦       asm lodsb                                                    ¦
¦       asm or      al,al                                            ¦
¦       asm jnz     ContinuaProcurando                               ¦
¦       asm mov     ax,cx                                            ¦
¦                                                                    ¦
¦#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__) ¦
¦       asm pop     ds          /* Restaura DS */                    ¦
¦#endif                                                              ¦
¦                                                                    ¦
¦       return _AX;                                                  ¦
¦   }                                                                ¦
+--------------------------------------------------------------------+
```

A rotina acima é equivalente a função strlen() de <string.h>.

Como disse antes, nos modelos COMPACT, LARGE e HUGE um pointer tem o formato SEGMENTO:OFFSET que é armazenado na memória em uma

grande variável de 32 bits (os 16 mais significativos são o SEGMENTO e os 16 menos significativos são o OFFSET). Nos modelos TINY, SMALL e MEDIUM apenas o OFFSET é fornecido no pointer (ele tem 16 bits neste caso), o SEGMENTO é o assumido em DS (não devemos alterá-lo, neste caso!).

Se você compilar essa listagem nos modelos COMPACT, LARGE ou HUGE o código coloca em DS:SI o pointer (lembre-se: pointer é só um outro nome para "endereço de memória!"). Senão, precisamos apenas colocar em SI o OFFSET (DS já está certo!).

Ao sair da função, DS deve ser o mesmo de antes da função ser chamada... Portanto, nos modelos "LARGOS" (hehe) precisamos salvar DS ANTES de usá-lo e restaura-lo DEPOIS de usado! O compilador não faz isso automaticamente!

Não se preocupe com SI (neste caso!)... este sim, o compilador salva sozinho...

Um macete com o uso de vetores pode ser mostrado no seguinte código exemplo:

```
char a[3];
int b[3], c[3];
long d[3];

void init(void)
{
    int i;

    for (i = 0; i < 3; i++)
        a[i] = b[i] = c[i] = d[i] = 0;
}
```

O compilador gera a seguinte função equivalente em assembly:

```
void init(void)
{
    asm xor     si,si           /* SI = i */
    asm jmp     short @1@98
@1@50:
    asm mov     bx,si           /* BX = i */
    asm shl     bx,1
    asm shl     bx,1            /* BX = BX * 4 */
    asm xor     ax,ax
    asm mov     word ptr [d+bx+2],0 /* ?! */
    asm mov     word ptr [d+bx],ax

    asm mov     bx,si
    asm shl     bx,1
    asm mov     [c+bx],ax

    asm mov     bx,si       /* ?! */
    asm shl     bx,1        /* ?! */
    asm mov     [b+bx],ax

    asm mov     [a+si],al
    asm inc     si
@1@98:
    asm cmp     si,3
    asm jl      short @1@50
}
```

Quando poderiamos ter:

```
void init(void)
{
    asm xor     si,si           /* SI = i = 0 */
    asm jmp     short @1@98
@1@50:
    asm mov     bx,si           /* BX = i */
    asm shl     bx,1
    asm shl     bx,1            /* BX = BX * 4 */
    asm xor     ax,ax           /* AX = 0 */
    asm mov     word ptr [d+bx+2],ax /* modificado! */
    asm mov     word ptr [d+bx],ax

    asm shr     bx,1            /* BX = BX / 2 */
    asm mov     [c+bx],ax
    asm mov     [b+bx],ax

    asm mov     [a+si],al
    asm inc     si
@1@98:
    asm cmp     si,3
    asm jl      short @1@50
}
```

Note que economizamos 3 instruções em assembly e ainda aceleramos um tiquinho, retirando o movimento de um valor imediato para memória (o 0 de "mov word ptr [d+bx+2],0"), colocando em seu lugar o registrador AX, que foi zerado previamente.

Isso parece besteira neste código, e eu concordo... mas, e se tivessemos:

```
void init(void)
{
    for (i = 0; i < 32000; i++)
        a[i] = b[i] = c[i] = d[i] =
        e[i] = f[i] = g[i] = h[i] =
        I[i] = j[i] = k[i] = l[i] =
        m[i] = n[i] = o[i] = p[i] =
        r[i] = s[i] = t[i] = u[i] =
        v[i] = x[i] = y[i] = z[i] =
        /* ... mais um monte de membros de vetores... */
        = _XYZ[i] = 0;
}
```

A perda de eficiência e o ganho de tamanho do código seriam enormes por causa da quantidade de vezes que o loop é exeutado (32000) e por causa do numero de movimentos de valores imediatos para memória, "SHL"s e "MOV BX,SI" que teriamos! Conclusão: Em alguns casos é mais conveniente manipular VARIOS vetores com funções escritas em assembly...

EXEMPLO de codificação: ** O swap() aditivado :)

Alguns códigos em C que precisam trocar o conteúdo de uma variável pelo de outra usam o seguinte macro:

```
#define swap(a,b) { int t; t = a; a = b; b = t; }
```

Bem... a macro acima funciona perfeitamente bem, mas vamos dar uma olhada no código assembly gerado pelo compilador pro seguinte programinha usando o macro swap():

```
#define swap(a,b) { int t; t = a; a = b; b = t; }

int x = 1, y = 2;

void main(void)
{ swap(x,y); }
```

O código equivalente, após ser pre-processado, ficaria:

```
int x = 2, y = 1;
void main(void) {
    int t;

    asm mov ax,x
    asm mov t,ax
    asm mov ax,y
    asm mov x,ax
    asm mov ax,t
    asm mov y,ax
}
```

No máximo, o compilador usa o registrador SI ou DI como variavel 't'... Poderiamos fazer:

```
int x = 2, y = 1;
void main(void)
{
    asm mov     ax,x
    asm mov     bx,y
    asm xchg    ax,bx
    asm mov     x,ax
    asm mov     y,ax
}
```

Repare que eliminamos uma instrução em assembly, eliminando também um acesso à memória e uma variável local... Tá bom... pode me chamar de chato, mas eu ADORO diminuir o tamanho e aumentar a velocidade de meus programas usando esse tipo de artifício! :)

ASSEMBLY XIV

Aqui estou eu novamente!!! Nos textos de "SoundBlaster Programming" a gente vai precisar entender um pouquinho sobre o TURBO ASSEMBLER, então é disso que vou tratar aqui, ok?

Well... O TURBO ASSEMBLER 'compila' arquivos .ASM, transformando-os em .OBJ (sorry "C"zeiros, mas os "PASCAL"zeiros talvez não estejam familiarizados com isso!). Os arquivos .OBJ devem ser linkados com os demais módulos para formar o arquivo .EXE final. Precisamos então conhecer como criar um .OBJ que possa ser linkado com códigos em "C" e "PASCAL". Eis um exemplo de um módulo em ASSEMBLY compatível com as duas linguagens:

```
IDEAL               ; Poe TASM no modo IDEAL
MODEL LARGE,PASCAL  ; Modelo de memória...
LOCALS
JUMPS

GLOBAL  ZeraAX : PROC   ; ZeraAX é público aos outros módulos

CODESEG     ; Inicio do (segmento de) código

PROC    ZeraAX          ; Inicio de um PROCedimento.
        sub     ax,ax
        ret
ENDP                    ; Fim do PROCedimento.

END     ; Fim do módulo .ASM
```

As duas linhas iniciais informam ao TURBO ASSEMBLER o modo de operação (IDEAL), o modelamento de memória (LARGE - veja discussão abaixo!) e o método de passagem de parametros para uma função (PASCAL).

O modo IDEAL é um dos estilos de programação que o TURBO ASSEMBLER suporta (o outro é o modo MASM), e é o meu preferido por um certo númeroo de razões. O modelo LARGE e a parametrização PASCAL também são minhas preferidas porque no modelo LARGE é possível termos mais de um segmento de dados e de código (podemos criar programas realmente GRANDES e com MUITA informação a ser manipulada!). PASCAL deixa o código mais limpo com relação ao conteúdo dos registradores após o retorno de uma função (alguns compiladores C, em algumas circunstancias, têm a mania de modificar o conteúdo de CX no retorno!). Fora isso PASCAL também limpa a pilha ANTES do retorno da procedure/função. Mas, isso tudo tem uma pequena desvantagem: Usando-se PASCAL, não podemos passar um númeroo variável de parametros pela pilha (os três pontos da declaração de uma função C: void f(char *, ...);)!

Ahhh... Você deve estar se perguntando o que é o LOCALS e JUMPS. LOCALS diz ao compilador que qualquer label começado por @@ é local ao PROC atual (não é visivel em outros PROCs!)... Assim podemos usar labels com mesmo nome dentro de várias PROCs, sem causar nenhuma confusão:

```
; modelamento, modo, etc...
LOCALS
```

```
PROC    F1
        mov cx,1000
@@Loop1:
        dec cx
        jnz @@Loop1
        ret
ENDP

PROC    F2
        mov cx,3000
@@Loop1:
        dec cx
        jnz @@Loop1
        ret
ENDP
;... O resto...
```

Repare que F1 e F2 usam o mesmo label (@@Loop1), mas o fato da diretiva LOCALS estar presente informa ao assembler que elas são diferentes!

Já JUMPS resolve alguns problemas para nós: Os saltos condicionais (JZ, JNZ, JC, JS, etc..) são relativos a posição atual (tipo: salte para frente tantas posições a partir de onde está!)... Em alguns casos isso pode causar alguns erros de compilação pelo fato do salto não poder ser efetuado na faixa que queremos... ai entra o JUMPS... Ele resolve isso alterando o código para que um salto incondicional seja efetuado. Em exmplo: Suponha que o label @@Loop2 esteja muito longe do ponto atual e o salto abaixo não possa ser efetuado:

```
    JNZ     @@Loop2
```

O assembler substitui, caso JUMPS esteja presente, por:

```
    JZ      @@P1
    JMP     @@Loop2     ; Salto absoluto se NZ!
@@P1:
```

A linha seguinte do exemplo inicial informa ao assembler que o PROCedimento ZeraAX é público, ou GLOBAL (visível por qualquer um dos módulos que o queira!). Logo após, a diretiva CODESEG informa o inicio de um segmento de código.

Entre as diretivas PROC e ENDP vem o corpo de uma rotina em assembly. PROC precisa apenas do nome da função (ou PROCedimento). Mais detalhes sobre PROC abaixo.

Finalizamos a listagem com END, marcando o fim do módulo em .ASM.

Simples, né?! Suponha agora que você queira passar um parametro para um PROC. Por exemplo:

```
; Equivalente a:
;   void pascal SetAX(unsigned v) { _AX = v; }
;   PROCEDURE SetAX(V:WORD) BEGIN regAX := V; END;
IDEAL
```

```
MODEL LARGE,PASCAL
LOCALS
JUMPS

GLOBAL SetAX : PROC

PROC    SetAX
ARG     V : WORD
        mov     ax,[V]
        ret
ENDP

END
```

Hummmm... Surgiu uma diretiva nova. ARG especifica a lista de parâmetros que deverá estar na pilha após a chamada de SetAX (ARGumentos de SetAX). Note que V está entre colchetes na instrução 'mov'... isso porque V é, na verdade, uma referência à memória (na pilha!) e toda referência à memória precisa ser cercada com colchetes (senão dá um baita erro de sintaxe no modo IDEAL!). Depois da compilação o assembler substitui V pela referência certa.

Os tipos, básicos, válidos para o assembler são: BYTE, WORD, DWORD... Não existe INTEGER, CHAR como em PASCAL (INTEGER = WORD com sinal; assim como CHAR = BYTE com sinal!).

Para finalizar: Em um único módulo podem existir vários PROCs:

```
IDEAL               ; modo IDEAL do TASM
MODEL LARGE, PASCAL ; modelamento de memória...
LOCALS
JUMPS

; ... aqui entra os GLOBALS para os PROCs que vc queira que
;     sejam públicos!

CODESEG     ; Começo do segmento de código...

PROC    P1
    ; ... Corpo do PROC P1
ENDP

PROC    P2
    ; ... Corpo do PROC P2
ENDP

;... outros PROCs...

END     ; Fim da listagem
```

Existem MUITOS outros detalhes com relação do TASM... mas meu objetivo no curso de ASM é a mixagem de código... pls, alguma dúvida, mandem mensagem para cá ou via netmail p/ mim em 12:2270/1.

ASSEMBLY XV

Continuando o papo sobre o TASM, precisaremos aprender como manipular tipos de dados mais complexos do que WORD, BYTE ou DWORD. Eis a descrição das estruturas!

Uma estrutura é o agrupamento de tipos de dados simples em uma única classe de armazenamento, por exemplo:

```
STRUC   MyType
    A   DB  ?
    B   DW  ?
ENDS
```

A estrutura MyType acima, delimitada pelas palavras-chava STRUC e ENDS, foi construida com dois tipos de dados simples (BYTE e WORD) com os nomes de A e B. Note que as linhas acima apenas declaram a estrutura, sem alocar espaço na memória para ela. Criar uma 'instancia' dessa estrutura é tão simples quanto criar uma variável de tipo simples:

```
MyVar   MyType  <0,0>
```

A sintaxe é basicamente a mesma de qualquer declaração de variável em assembly, com a diferença de que o 'tipo' do dado é o nome (ou TAG) da estrutura - MyType - e os dados iniciais dos elementos da estrutura estão localizados entre os simbolos < e >. Na linha acima criamos a variável MyVar, cujos elementos são 0 e 0. Vamos a um exemplo de uso desse novo tipo:

```
;... Aqui entra o modelamento,...

DATASEG

MyVar   MyType  <0,0>

CODESEG

PROC    SetA        ; Poe valor em A na estrutura.
ARG     V : Byte
        mov     al,[V]
        mov     [MyVar.A],al
        ret
ENDP

PROC    SetB        ; Poe valor em B na estrutura.
ARG     V : Word
        mov     ax,[V]
        mov     [MyVar.B],ax
        ret
ENDP

;... Aqui entra o fim do código...
```

Simples, não?

Mas, e se quisermos trabalhar com um vetor do tipo MyType? Vetores de tipos mais simples é facil:

```
DATASEG

MyVar1  dw  10 DUP (0)

CODESEG

PROC    Fill1
    mov     cx,10
    sub     bx,bx
@@FillType1:
    mov     [bx+MyVar1],0FFh
    add     bx,2
    dec     cx
    jnz     @@FillType1
    ret
ENDP
```

Aqui fiz da maneira mais dificil apenas para exemplificar um método de preenchimento de vetores. No caso, BX contém o item desejado do vetor. MyVar1 é o deslocamento do primeiro item do vetor na memória e CX a quantidade de itens do vetor. Note que temos um vetor de WORDS e precisaremos adicionar 2 (tamnho de uma WORD) para cara item do vetor. No caso da estrutura, isso fica um pouco mais complicado porque ela pode ter um tamanho não múltiplo de 2 (o que complica o cálculo. Por exemplo, MyType (a estrutura) tem 3 bytes de tamanho. Eis a implementação (não otimizada) para a rotina FillType para preenchimento de um vetor de MyType com 10 itens:

```
DATASEG
MyVar   MyType  10 dup (<0,0>)

CODESEG
PROC    FillType
        mov     cx,10
        sub     bx,bx   ; indice para localizar itens.
@@FillLoop:
        mov     [bx+MyVar.A],0FFh   ; * Instrução destacada...
        mov     [bx+MyVar.B],0FFFFh
        add     bx,3
        dec     cx
        jnz     @@FillLoop
        ret
ENDP
```

Essa rotina merece ser observada mais de perto:

Vejamos a instrução destacada na listagem acima... MyVar.A fornece o deslocamento de A, do primeiro item do vetor, na memória, enquanto isso BX fornece o indice do item desejado no vetor. Assim, BX+MyVar.A fornecerá o offset do elemento A do item da estrutura desejado.

Well... É isso...

+--------------+
¦ ASSEMBLY XVI ¦
+--------------+

Usando a memória Expandida (EMS).

Muitos modplayers hoje em dia usam a memória expandida para armazenar os samples. Neste texto veremos como funciona a memória expandida e como usá-la...

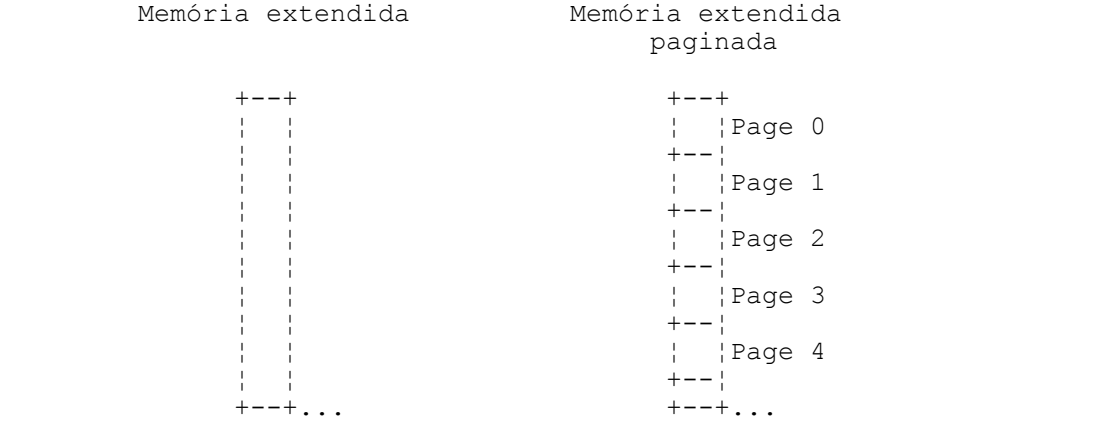
A maioria dos PC-ATs com mais de 1Mb de memória possui dois tipos de memória:

¦ Convencional - Na faixa de 0 até 1Mb
¦ Extendida: de 1Mb em diante.

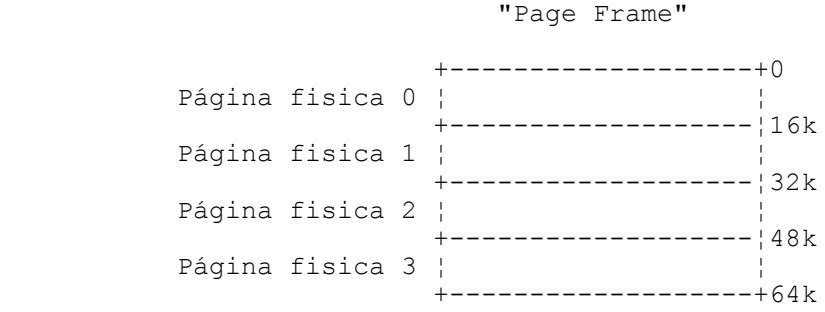
A memória extendida é facilmente manipulável quando um programa está em modo protegido e com toda a memória mapeada em um ou mais seletores. Os 386s permitem que um seletor acesse um segmento de até 4Gb de tamanho... Mas, não é esse o nosso caso. Temos um pequeno programa rodando sob o MS-DOS, no modo real (modo nativo dos processadores Intel), que tem acesso somente à memória convencional. Podemos acessar a memória extendida através do driver HIMEM.SYS ou usando uma função de movimento de blocos da BIOS, mas isso aumentaria em muito a complexidade do software (e, por conseguência, seu tamanho).

A Lotus, Intel e Microsoft criaram a especificação EMS para esse caso. O programa EMM386.EXE, ou qualquer outro gerenciador de memória como o QEMM, emula a memória expandida da mesma forma que uma máquina com apenas este tipo de memória faria (A memória expandida por hardware não fez muito sucesso nos EUA como a memória extendida!). A especificação EMS simplesmente usa um espaço da memória convencional (chamado de "Page Frame") para armazenar "páginas" de 16kb da memória extendida. Isto é... divida a sua memória extendida em diversos blocos de 16k e terá o númeroo de páginas (pages) que poderão estar disponíveis para uso.

O EMM (Expanded Memory Manager) simplesmente faz a cópia das páginas desejadas para o "Page Frame" para que o nosso software posssa lê-las e escrevê-las, copiando-as de volta para as páginas corretas quando fizermos a troca de páginas do "Page Frame". No "Page Frame" cabem, normalmente, 4 páginas... fazendo um total de 64kb (ou seja, exatamente o tamanho de um segmento!). Considere a figura abaixo:



Ok... a memória extendida foi dividida em 'n' páginas de 16k. O "Page Frame" fica na memória convencional. Por exemplo, suponha que o "Page Frame" esteja localizado no segmento 0C000h:



Do offset 0 até 16k-1 fica a primeira página do "Page Frame", do offset 16k até 32k-1 a segunda, e assim por diante. A especificação EMS nos permite colocar apenas 4 páginas no "Page Frame". Assim, o nosso programa escolhe cada uma das quatro "páginas lógicas" que serão copiadas da memória extendida para cada uma das quatro "páginas fisicas" do Page Frame.

Vale a pena lembrar que o Page Frame está sempre em algum lugar da memória convencional, portanto acessível aos programas feitos para MS-DOS, que normalmente trabalham em modo real.

A interrupção 67h é a porta de entrada para as funções do EMM (EMM386, QEMM, 386MAX, entre outros). Mas antes de começarmos a futucar o EMM precisamos saber se ele está presente... Eis a rotina de detecção do EMM p/ os compiladores C da BORLAND:

```
---%<-----------------------------------%<---------------------------
#include <io.h>
#include <fcntl.h>
#include <dos.h>

#define CARRY_BIT   (_FLAGS & 0x01)

/* Obtém a maior versão do EMM - definida em outro módulo! */
extern int emm_majorVer(void);

/* Testa a presença do EMM
   Retorna 0 se EMM não presente ou versão < 3.xx
   Retorna 1 se tudo ok! */
int isEMMpresent(void)
{
    int handle;

    /* Tenta abrir o device driver EMMXXXX0 para leitura! */
    if ((handle = open("EMMXXXX0", O_BINARY | O_RDONLY)) == -1)
        return 0;   /* Não tem EMM! */

    /* Verifica se é um arquivo ou dispositivo
       Usa IOCTL para isso! */
    _BX = handle;
    _AX = 0x4400;
    geninterrupt(0x21);
    if (!(_DX & 0x80))
        return 0;   /* É um arquivo!!! Não é o EMM! */

    /* Verifica o dispositivo está ok */
```

```
    _BX = handle;
    _AX = 0x4407;
    geninterrupt(0x21);
    if (CARRY_BIT || !_AL) return 0; /* Não está ok */

    /* Verifica a versão do EMM.
       Para nossos propósitos tem que ser >= que
       3.xx */
    if (emm_majorVer() < 3) return 0; /* Não é ver >= 3.xx */

    /* Tudo ok... EMM presente */
    return 1;
}
---%<-----------------------------------%<---------------------------
```

No próximo texto mostrarei como usar o EMM.

See ya...

Eis o arquivo .ASM com as rotinas para manipulação da memória expandida:

```
IDEAL
MODEL LARGE,PASCAL
LOCALS
JUMPS

GLOBAL  emmGetVersion : PROC
GLOBAL  emmGetPageFrameSegment : PROC
GLOBAL  emmGetAvailablePages : PROC
GLOBAL  emmAllocPages : PROC
GLOBAL  emmFreePages : PROC
GLOBAL  emmMapPage : PROC
GLOBAL  emmGetError : PROC

DATASEG

emmVersion  dw  0
emmError    db  0       ; Nenhum erro ainda... :)

CODESEG

; Obtém a versão do EMM.
; Devolve no formato 0x0X0Y (onde X é versão e Y revisão).
; Protótipo em C:
;   unsigned pascal emmGetVersion(void);
PROC    emmGetVersion
    mov     [emmError],0    ; Inicializa flag de erro...
    mov     ah,46h
    int     67h             ; Invoca o EMM
    or      ah,ah           ; Testa o sucesso da função...
    jz      @@no_error
    mov     [emmError],ah   ; Poe erro no flag...
    mov     ax,-1           ; ... e retorna != 0.
    jmp     @@done
    mov     ah,al           ; Prepara formato da versão.
    and     ax,111100001111b ; A função 46h do EMM devolve
    mov     [emmVersion],ax  ; no formato BCD... por isso
@@done:                      ; precisamos formatar...
    ret
ENDP

; Função: Obtém o segmento do Page Frame.
; Protótipo em C:
;   unsigned pascal emmGetPageFrameSegment(void);
PROC    emmGetPageFrameSegment
    mov     ah,41h      ; Usa a função 41h do EMM
    int     67h         ; Chama o EMM
    mov     ax,bx       ; Poe o segmento em AX
                        ; Função 41h coloca o segmento do
                        ; "Page Frame" em BX.
    ret
ENDP

; Função: Obtém o númeroo de páginas disponíveis na memória.
; Protótipo em C:
;   unsigned pascal emmGetAvailablePages(void);
; Obs:
```

```
;   Não verifica a ocorrencia de erros... modifique se quiser
PROC    emmGetAvailablePages
    mov     ah,42h
    int     67h     ; Invoca o EMM.
    mov     ax,bx   ; Poe páginas disponiveis em AX.
    ret
ENDP

; Aloca páginas e devolve handle.
; Protótipo em C:
;   int pascal emmGetAvailablePages(unsigned Pages);
; Obs: Devolve -1 se houve erro na alocaçao e seta
;      a variável emmError.
PROC    emmAllocPages
ARG     Pages:WORD
    mov     [emmError],0    ; Inicializa flag de erros...
    mov     bx,[Pages]      ; BX = númeroo de páginas a alocar
    mov     ah,43h
    int     67h             ; Invoca o EMM.
    or      ah,ah           ; Verifica erro do EMM.
    jz      @@no_error
    mov     [emmError],ah   ; Poe erro na variável emmError
    mov     dx,-1
@@no_error:
    mov     ax,dx           ; retorna código de erro.
                            ; ou o handle.
    ret
ENDP

; Libera páginas alocadas.
; Protótipo em C:
;   void pascal emmFreePages(int handle);
; Obs: Não verifica erros... modifique se quiser...
PROC    emmFreePages
ARG     handle:WORD
    mov     dx,[handle]
    mov     ah,45h
    int     67h
    ret
ENDP

; Mapeia uma página no Page Frame.
; Protótipo em C:
;   int pascal emmMapPage(int handle,
;                         unsigned char pfPage,
;                         unsignec PageNbr);
; Onde: handle é o valor devolvido pela função de alocação de
;       páginas.
;       pfPage é o númeroo da página do Page Frame (0 até 3).
;       PageNbr  é o númeroo da página a ser colocada no
;       Page Frame (0 até máximo - 1).
; Devolve -1 se ocorreu erro e seta a variável emmError.
PROC    emmMapPage
ARG     handle:WORD, pfPage:BYTE, PageNbr:WORD
    mov     [emmError],0
    mov     ah,44h
    mov     al,[pfPage]
    mov     bx,[PageNbr]
    mov     dx,[handle]
    int     67h
    or      ah,ah
    jz      @@no_error
    mov     [emmError],ah
    mov     ah,-1
```

```
  @@no_error:
      mov     al,ah
      ret
  ENDP

  ; Retorna com o erro do EMM.
  ; Protótipo:
  ;   int pascal emmGetError(void);
  PROC    emmGetError
      mov     ax,[emmError]
      ret
  ENDP

  END
```

Esta é uma implementação simplificada, mas para nossos propósitos serve muito bem. Algumas considerações: A alocação de memória via EMM não é feita da mesma maneira que a função malloc() de C ou GetMem() do TURBO PASCAL. Não é devolvido nenhum pointer. Isto se torna óbvio a partir do momento que entendemos como funciona o EMM: Toda a manipulação de bancos de memória é feita de forma indireta pelo Page Frame. A função de alocação deve apenas devolver um handle para que possamos manipular as páginas alocadas. Entenda esse handle da mesma forma com que os arquivos são manipulados... Se quisermos usar um banco alocado precisamos informar ao EMM qual dos bancos queremos usar, fazendo isso via o handle devolvido pelo próprio EMM.

Suponha que queiramos alocar 128kb da memória expandida para o nosso programa. Precisamos alocar 8 páginas lógicas (8 * 16k = 128k). Chamariamos a função emmAllocPages() em C da seguinte forma:

```
  #include <conio.h>
  #include <stdlib.h>

  int emm_handle;

  void f(void)
  {
      /* ... */
      if ((emm_handle = emmAllocPages(8)) == -1) {
          cprintf("EMM ERROR #%d\r\n", emmGetError());
          exit(1);
      }
      /* ... */
  }
```

Na função emmAllocPages() optei por devolver -1 para indicar o insucesso da função... Você pode arrumar um esquema diferente para chegar isso (por exemplo, checando a variável emmError após a chamada a função!).

Well... Temos 8 páginas lógicas disponíveis. E agora?... As 8 páginas estão sempre numeradas de 0 até o máximo - 1. No nosso caso teremos as páginas 0 até 7 disponíveis ao nosso programa. Lembre-se que cada uma tem apenas 16k de tamanho e que podem ser arranjadas de qq maneira q vc queira no Page Frame. Vamos usar as 4 páginas iniciais como exemplo... para isso precisamos mapea-las no Page Frame usando a função emmMapPage().

```
void f(void)
{
    int i;

    /* ... */
    for (i = 0; i < 4; i++)
        emmMapPage(emm_handle,i,i);
}
```

Depois deste pequeno loop sabemos que qualquer alteração no conteúdo do Page Frame alterará as páginas que estão mapeadas nele...:) Simples né? Só nos resta conhecer o endereço inicial do Page Frame:

```
#include <dos.h>

void far *PageFrameAddr;

void f(void)
{
    /* ... */
    PageFrameAddr = MK_FP(emmGetPageFrameSegment(), 0);
    /* ... */
}
```

Ao fim do uso da memória expandida precisamos dealocar o espaço previamente alocado... C e C++ dealocam automaticamente qualquer espaço alocado por malloc(), calloc() e funções afins... Não é o caso de nossas rotinas acima... então acostume-se a manter a casa em ordem e usar a função emmFree() quando não precisar mais das páginas alocadas.

Isso tudo não funcionará se o EMM não estiver instalado... No texto anterior mostrei a rotina para determinar a presença do EMM. E, no mesmo texto, apareceu a rotina emm_majorVer(). Eis a rotina abaixo:

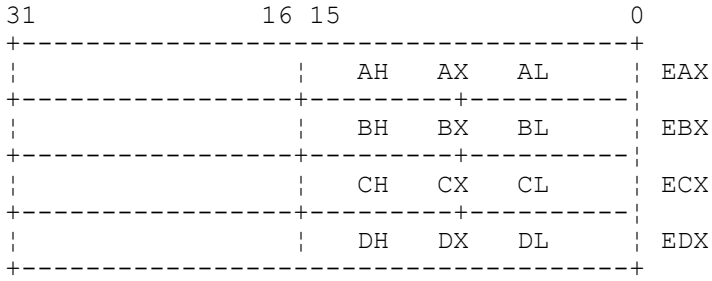
```
int emm_majorVer(void)
{ return ((int)emmGetVersion() >> 8); }
```

See ya l8tr

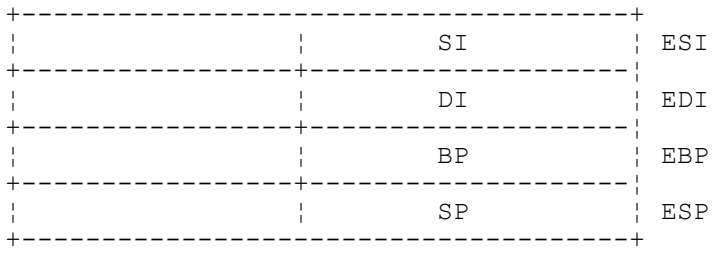
```
+----------------+
¦ ASSEMBLY XVIII ¦
+----------------+
```

Hummmm... Estamos na era dos 32 bits... então por que esperar mais para discutirmos as novidades da linha 386 e 486? Eles não diferem muito do irmão menor: o 8086. A não ser pelo fato de serem "maiores". :)

O 8086 e 80286 têm barramento de dados de 16 bits de tamanho enquanto o 386 e o 486 tem de 32 bits. Nada mais justo que existam modificações nos registradores também:



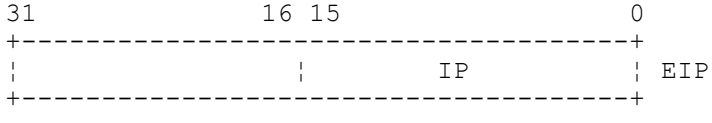
Os registradores de uso geral continuam os velhos conhecidos de sempre... Só que existem os registradores de uso geral de 32 bits: EAX, EBX, ECX e EDX, onde os 16 bits menos significativos destes são AX, BX, CX e DX, respectivamente.



Da mesma forma, os registradores SI, DI, BP e SP ainda estão aqui... bem como os seus equivalentes de 32 bits: ESI, EDI, EBP e ESP.

Os registradores de segmento (chamados de SELETORES desde o surgimento do 80286) são os mesmos e não mudaram de tamanho, continuam com 16 bits: CS, DS, ES e SS. Mas acrecentaram outros: FS e GS. Isto é... Agora existe um registrador de segmento de código (CS), um segmento de pilha (SS) e quatro segmentos de dados (DS, ES, FS e GS). Lembrando que DS é o segmento de dados default. Repare na ordem alfabética dos registradores de segmento de dados...

O registrador Instruction Pointer também continua o mesmo... E também existe o seu irmão maior... EIP:



Da mesma forma os FLAGS também são os mesmos de sempre... mas o registrador FLAGS também foi expandido para 32 bits e chamado de EFLAGS. Os sinalizadores extras são usados em aplicações especiais (como por exemplo, chaveamento para modo protegido, modo virtual,

chaveamento de tarefas, etc...).

Alguns outros registradores foram adicionados ao conjunto: CR0, CR1, CR3, TR4 a TR7. DR0 a DR3, DR6 e DR7 (todos de 32 bits de tamanho). Esses novos registradores são usados no controle da CPU (CR?), em testes (TR?) e DEBUG (DR?). Não tenho maiores informações sobre alguns deles e por isso não vou descrevê-los aqui.

Novas instruções foram criadas para o 386 e ainda outras mais novas para o 486 (imagino que devam existir outras instruções específicas para o Pentium!). Eis algumas delas:

¦ BSF (Bit Scan Forward)

Processador: 386 ou superior

Sintaxe: BSF dest,src

Descrição:

Procura pelo primeiro bit setado no operando "src". Se encontrar, coloca o numero do bit no operando "dest" e seta o flag Zero. Se não encontrar, não altera o operando "dest" e reseta o flag Zero. BSF procura o bit setado começando pelo bit 0 do operando "src".

Exemplo:

```
BSF     AX,BX
```

¦ BSR (Bit Scan Reverse)

Processador: 386 ou superior

Sintaxe: BSR dest,src

Descrição:

Faz a mesma coisa que BSF, porém a ordem de procura começa a partir do bit mais significativo do operando "src".

Exemplo:

```
BSR     AX,BX
```

¦ BSWAP

Processador: 486 ou superior

Sintaxe: BSWAP reg32

Descrição:

Inverte a ordem das words de um registrador de 32 bits.

Exemplo:

```
BSWAP EAX
```

¦ BT (Bit Test)

Processador: 386 ou superior

Sintaxe: BT dest,src

Descrição:

Copia o conteúdo do bit do operando "dest" indicado pelo operando "src" para o flag Carry.

Exemplo:

```
BT  AX,3
```

Observações:

1- Aparentemente esta instrução não aceita operandos de 32 bits.
2- No exemplo acima o bit 3 de AX será copiado para o flag Carry.

¦ BTC (Bit Test And Complement)

Processador: 386 ou superior

Sintaxe: BTC dest,src

Descrição:

Instrução identica à BT, porém complementa (inverte) o bit do operando "dest".

¦ BTR e BTS

Processador: 386 ou superior

Sintaxe:
```
BTR dest,src
BTS dest,src
```

Descrição:

Instruções identicas a BT, porém BTR zera o bit do operando destino e BTS seta o bit do operando destino.

¦ CDQ (Convert DoubleWord to QuadWord)

Processador: 386 ou superior

Sintaxe: CDQ

Descrição:

Expande o conteúdo do registrador EAX para o par EDX e EAX, preenchendo com o bit 31 de EAX os bits de EDX (extensão de sinal).

¦ CWDE (Convert Word to DoubleWord Extended)

Processador: 386 ou superior

Sintaxe: CWDE

Descrição:

Esta instrução expande o registrador AX para EAX, considerando o sinal. Ela é equivalente a instrução CWD, porém não usa o par DX:AX para isso.

¦ CMPXCHG

Processador: 486 ou superior

Sintaxe: CMPXCHG dest,src

Descrição:

Compara o acumulador (AL, AX ou EAX - dependendo dos operandos) com o operando "dest". Se forem iguais o acumulador é carregado com o conteúdo de "dest", caso contrário com o conteúdo de "src".

Exemplo:

```
CMPXCHG BX,CX
```

¦ INVD (Invalidate Cache)

Processador: 486 ou superior

Sintaxe: INVD

Descrição:

Limpa o cache interno do processador.

¦ JECXZ

Processador: 386 ou superior

Observação: É identica a instrução JCXZ, porém o teste é feito no registrador extendido ECX (32 bits).

¦ LGS e LFS

Processador: 386 ou superior

Observação: Essas instruções são identicas as instruções LDS e LES, porém trabalham com os novos registradores de segmento.

¦ MOVSX e MOVZX

Processador: 386 ou superior

Sintaxe: MOVSX dest,src
MOVZX dest,src

Descrição:

Instruções úteis quando queremos lidar com operandos de tamanhos diferentes. MOVZX move o conteúdo do operando "src" para "dest" (sendo que "src" deve ser menor que "dest") zerando os bits extras. MOVSX faz a mesma coisa, porém copiando o último bit de "src" nos bits extras de "dest" (conversão com sinal).

Exemplo:

* Usando instruções do 8086, para copiar AL para BX precisariamos fazer isto:

```
MOV     BL,AL
MOV     BH,0
```

* Usando MOVZX podemos simplesmente fazer:

```
                MOVZX   BX,AL
```

¦ Instrução condicional SET

Processador: 386 ou superior

Sintaxe: SET? dest
(Onde ? é a condição...)

Descrição:

Poe 1 no operando destino se a condição for satisfeita. Caso contrário poe 0.

Exemplo:

```
SETNZ AX
SETS  EBX
SETZ  CL
```

¦ SHRD e SHLD (Double Precision Shift)

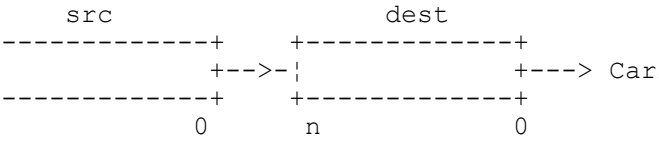
Processador: 386 ou superior

Sintaxe: SHRD dest,src,count
SHLD dest,src,count

Descrição:

Faz o shift para esquerda (SHLD) ou direita (SHRD) do operando "dest" "count" vezes, porém os bits que seriam preenchidos com zeros são preenchidos com o contéudo dos bits do operando "src". Eis um gráfico exemplificando:

SHRD



O operando "src" não é alterado no processo. O flag de Carry contém o último bit que "saiu" do operando "dest".

Exemplo:

```
SHLD    EAX,ECX,3
SHRD    AX,BX,CL
```

¦ Instruções que manipulam blocos...

CMPSD, LODSD, MOVSD, STOSD, INSD e OUTSD se comportam da mesma forma que suas similares de 8 ou 16 bits (CMPSB, CMPSW, etc..), porém usam os registradores extendidos (ESI, EDI, ECX, EAX) e operam com dados de 32 bits de tamanho (DoubleWords).

Existem mais instruções... Consulte algum manual da Intel ou o hipertexto HELPPC21... Pedirei aos Sysops do VixNET BBS (agora com 6 linhas hehehe) para deixarem disponivel o arquivo 386INTEL.ZIP... que é o guia técnico para o processador 386.

Dúvidas a respeito dos novos recursos:

¦Q¦ Os segmentos tem mais que 64k no modo real, já que os registradores extendidos podem ser usados neste modo? Como funcionaria uma instrução do tipo:

```
    MOV     [ESI+3],EAX
```

¦R¦ Não... no modo real os segmentos continuam a ter 64k de tamanho. Os registradores extendidos podem ser usados a vontade e, quando usados como offset em um segmento, os 16 bits superiores são ignorados. A instrução apresentada funcionaria da mesma forma que:

```
    MOV     [SI+3],EAX
```

¦Q¦ Onde e quando deve-se usar os novos registradores de segmentos?
¦R¦ Onde e quando você quiser. Pense neles como se fosse novos segmentos de dados extras. Na realidade você apenas conseguirá usá-los se explicitá-los numa instrução que faz referência à memória, por exemplo:

```
    MOV FS:[BX],AL
```

¦Q¦ Posso usar os registradores extendidos nas instruções normais ou apenas nas novas instruções?
¦R¦ Pode usá-los nas instruções "normais". A não ser que a instrução não permita operandos de 32 bits...

That's all for now, pals...

ASSEMBLY XIX

Oi povo...

Estou retomando o desenvolvimento do curso de assembly aos poucos e na nova série: Otimização de código para programadores C. Well... vão algumas das rotinas para aumentar a velocidade dos programas C que lidam com strings:

strlen()

A rotina strlen() é implementada da seguinte maneira nos compiladores C mais famosos:

```
int strlen(const char *s)
{
    int i = 0;
    while (*s++) ++i;
    return i;
}
```

Isso gera um código aproximadamente equivalente, no modelo small, a:

```
PROC    _strlen NEAR
ARG     s:PTR
    push    si                  ; precisamos preservar
    push    di                  ;  SI e DI.
    xor     di,di               ; i = 0;
    mov     si,s
@@_strlen_loop:
    mov     al,[si]
    or      al,al               ; *s == '\0'?
    jz      @@_strlen_exit      ; sim... fim da rotina.
    inc     si                  ; s++;
    inc     di                  ; ++i;
    jmp     short @@_strlen_loop ; retorna ao loop.
@@_strlen_exit:
    mov     ax,si               ; coloca i em ax.
    pop     si                  ; recupara SI e DI.
    pop     di
    ret
ENDP
```

Eis uma implementação mais eficaz:

```
#ifdef __TURBOC__
#include <dos.h>      /* Inclui pseudo_registradores */
#define _asm  asm
#endif

int     Strlen(const char *s)
{
    _asm push    es
```

```
 #ifndef __TURBOC__
       _asm push    di
 #endif

 #if defined(__LARGE__) || defined(__HUGE__) || defined(__COMPACT__)
       _asm les     di,s
 #else
       _asm mov     di,ds
       _asm mov     es,di
       _asm mov     di,s
 #endif

       _asm mov     cx,-1
       _asm sub     al,al
       _asm repne   scasb

       _asm not     cx
       _asm dec     cx
       _asm mov     ax,cx

 #ifndef __TURBOC__
       _asm pop     di
 #endif

       _asm pop     es

 #ifdef __TURBOC__
      return _AX;
 #endif
   }
```

Essa nova Strlen() [Note que é Strlen() e não strlen(), para não confundir com a função que já existe na biblioteca padrão!] é, com certeza, mais rápida que strlen(), pois usa a instrução "repne scasb" para varrer o vetor a procura de um caracter '\0', ao invés de recorrer a várias instruções em um loop. Inicialmente, CX tem que ter o maior valor possível (-1 não sinalizado = 65535). Essa função falha no caso de strings muito longas (maiores que 65535 bytes), dai precisaremos usar strlen()!

Uma vez encontrado o caracter '\0' devemos inverter CX. Note que se invertermos 65535 obteremos 0. Acontece que o caracter '\0' tambem é contado... dai, depois de invertermos CX, devemos decrementá-lo também, excluindo o caracter nulo!

Não se preocupe com DI se vc usa algum compilador da BORLAND, o compilador trata de salvá-lo e recuperá-lo sozinho...

strcpy()

Embora alguns compiladores sejam espertos o suficiente para usar as intruções de manipulação de blocos a implementação mais comum de strcpy é:

```
 char *strcpy(char *dest, const char *src)
 {
     char *ptr = dest;
     while (*dest++ = *src++);
     return ptr;
 }
```

Para maior compreenção a linha:

```
    while (*dest++ = *src++);
```

Pode ser expandida para:

```
    while ((*dest++ = *src++) != '\0');
```

O código gerado, no modelo small, se assemelha a:

```
PROC    _strcpy
ARG     dest:PTR, src:PTR
    push    si          ; Salva SI e DI
    push    di

    mov     si,[dest]  ; Carrega os pointers

    push    si                  ; salva o pointer dest

    mov     di,[src]

@@_strcpy_loop:
    mov     al,byte ptr [di]    ; Faz *dest = *src;
    mov     byte ptr [si],al

    inc     di                  ; Incrementa os pointers
    inc     si

    or      al,al               ; AL == 0?!
    jne     short @@_strcpy_loop ; Não! Continua no loop!

    pop     ax                  ; Devolve o pointer dest.

    pop     di          ; Recupera DI e SI
    pop     si

    ret
ENDP
```

Este código foi gerado num BORLAND C++ 4.02! Repare que as instruções:

```
    mov        al,byte ptr [di]    ; Faz *dest = *src;
    mov        byte ptr [si],al
```

Poderiam ser facilmente substituidas por um MOVSB se a ordem dos registradores de índice não estivesse trocada. Porém a substituição, neste caso, causaria mais mal do que bem. Num 386 as instruções MOVSB, MOVSW e MOVSD consomem cerca de 7 ciclos de máquina. No mesmo microprocessador, a instrução MOV, movendo de um registrador para a memória consome apenas 2 ciclos. Perderiamos 3 ciclos em cada iteração (2 MOVS = 4 ciclos). Numa string de 60000 bytes, perderiamos cerca de 180000 ciclos de máquina... Considere que cada ciclo de máquina NAO é cada ciclo de clock. Na realidade um único ciclo de máquina equivale a alguns ciclos de clock - vamos

pela média... 1 ciclo de máquina ¸ 2 ciclos de clock, no melhor dos casos!

Vamos dar uma olhada no mesmo código no modelo LARGE:

```
  PROC _strcpy
  ARG  dest:PTR, src:PTR
  LOCAL temp:PTR
      mov        dx,[word high dest]
      mov        ax,[word low dest]
      mov        [word high temp],dx
      mov        [word low temp],ax

  @@_strcpy_loop:
      les        bx,[src]

      inc        [word low src]

      mov        al,[es:bx]

      les        bx,[dest]

      inc        [word low dest]

      mov        [es:bx],al

      or         al,al
      jne        short @@_strcpy_loop

      mov        dx,[word high temp]
      mov        ax,[word low temp]
      ret
  _strcpy    endp
```

Opa... Cade os registradores DI e SI?! Os pointers são carregados varias vezes durante o loop!!! QUE DESPERDICIO! Essa strcpy() é uma séria candidata a otimização!

Eis a minha implementação para todos os modelos de memória (assim como Strlen()!):

```
   char *Strcpy(char *dest, const char *src)
   {
        _asm    push    es
 #if defined(__LARGE__) || defined(__HUGE__) || defined(__COMPACT__)
        _asm    push    ds
        _asm    lds     si,src
        _asm    les     di,dest
 #else
        _asm    mov     si,ds
        _asm    mov     es,si
        _asm    mov     si,src
        _asm    mov     di,dest
 #endif
        _asm    push    si

   Strcpy_loop:
        _asm    mov     al,[si]
        _asm    mov     es:[di],al

        _asm    inc     si
```

```
        _asm    inc     di

        _asm    or      al,al
        _asm    jne     Strcpy_loop

        _asm    pop     ax
 #if defined(__LARGE__) || defined(__HUGE__) || defined(__COMPACT__)
        _asm    mov     ax,ds
        _asm    mov     dx,ax
        _asm    pop     ds
 #endif
        _asm    pop     es
   }
```

Deste jeito os pointers são carregados somente uma vez, os registradores de segmento DS e ES são usados para conter as componentes dos segmentos dos pointers, que podem ter segmentos diferentes (no modelo large!), e os registradores SI e DI são usados como indices separados para cada pointer!

A parte critica do código é o interior do loop. A única diferença entre essa rotina e a rotina anterior (a não ser a carga dos pointers!) é a instrução:

```
        _asm    mov     es:[di],al
```

Que consome 4 ciclos de máquina. Poderiamos usar a instrução STOSB, mas esta consome 4 ciclos de máquina num 386 (porém 5 num 486). Num 486 a instrução MOV consome apenas 1 ciclo de máquina! Porque MOV consome 4 ciclos neste caso?! Por causa do registrador de segmento explicitado! Lembre-se que o registrador de segmento DS é usado como default a não ser que usemos os registradores BP ou SP como indice!

Se vc está curioso sobre temporização de instruções asm e otimização de código, consiga a mais nova versão do hypertexto HELP_PC. Ele é muito bom. Quanto a livros, ai vão dois:

¦ Zen and the art of assembly language
¦ Zen and the art of code optimization

Ambos de Michael Abrash.

AHHHHHHHH... Aos mais atenciosos e experientes: Não coloquei o prólogo e nem o epílogo das rotinas em ASM intencionalmente. Notem que estou usando o modo IDEAL do TURBO ASSEMBLY para não confundir mais ainda o pessoal com notações do tipo: [BP+2], [BP-6], e detalhes do tipo decremento do stack pointer para alocação de variáveis locais... Vou deixar a coisa o mais simples possível para todos...

Da mesma forma: Um aviso para os novatos... NAO TENTEM COMPILAR os códigos em ASM (Aqueles que começão por PROC)... Eles são apenas uma demonstração da maneira como as funções "C" são traduzidas para o assembly pelo compilador, ok?

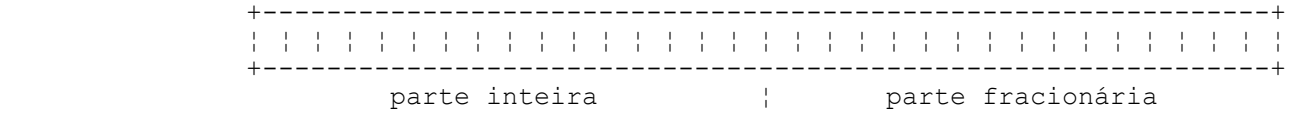
Well... próximo texto tem mais...

+-------------+
¦ ASSEMBLY XX ¦
+-------------+

Impressionante como as demonstrações gráficas (DEMOS) conseguem ser tão rápidas com todas aquelas transformações geométricas (objetos movimentando-se no espaço tridimensional), musicas em background, etc... A complexidade sugere a utilização de rotinas em ponto-flutuante para os calculos "cabeludos"... Opa! Ponto-flutuante?! Mas isso é muito lerdo!!!! Toma muito tempo de CPU... E nem sempre o feliz proprietário de um microcomputador tem um 486DX ou um 386 com co-processador! Como é que esses caras conseguem tanta velocidade?!

A resposta pode estar num método conhecido como "aritimética de ponto-fixo", que é o objetivo deste texto!

Imagine que possamos escrever um número "quebrado" (com casas decimais) da seguinte maneira:



A "casa" mais a esquerda é o bit mais significativo, e a mais a direita o menos significativo. Assim os 16 bits mais significativos (parte inteira) nos diz a "parte inteira" do númeroo (lógico, né?). E os 16 bits menos significativos (parte fracionária) nos diz a parte fracionária do númeroo (outra vez, lógico!). De forma que o bit menos significativo destes 32 bits é equivalente a 2 elevado a potência de -16 (ou seja: 1/65536). Eis um exemplo:

```
+-------------------------------------------------------------------+
¦  0000000000000000.1000000000000000b = 0.5 = 1/2                   ¦
¦  0000000000000000.0100000000000000b = 0.25 = 1/4                  ¦
¦  0000000000000000.0010000000000000b = 0.125 = 1/8                 ¦
¦  0000000000000000.1110000000000000b = 0.875                       ¦
¦  0000000000000001.1000000000000000b = 1.5 = 1 + 1/2               ¦
¦  0000000000000011.0010010000111111b =
(aprox.)                ¦
¦  0000000000000000.1101110110110011b ¸ cos(
/6) ¸ 0.866 (aprox.) ¦
+-------------------------------------------------------------------+
```

Não sei se deu para entender, mas do bit menos significativo até o mais significativo, o expoente vai aumentando, só que o bit menos significativo tem expoente -16. Assim, o bit 1 tem expoente -15, o seguinte -14, etc... até o último, 15. O ponto entre os dois conjuntos de 16 bits foi adicionado apenas para facilitar a visualização no exemplo acima.

Ok... então é possível representar "números quebrados" em dois conjuntos de 16 bits... a pergunta é: Pra que?!

Aritimética com números inteiros sempre é mais rápida do que a aritimética com números em ponto-flutuante. Tendo co-processador ou não! Mesmo que vc tenha um 486DX4 100MHz, os calculos em ponto-flutuante serão mais lerdamente efetuados do que os mesmos calculos com números inteiros (usando os registradores da CPU!). Neste ponto entra a aritimética de ponto-fixo (note que o "ponto decimal" não muda de posição...). Vejamos o que acontece se somarmos dois números em ponto fixo:

```
+-----------------------------------------------------------------+
|  0.25 + 1.75 = 2.0                                              |
|                                                                 |
|    0000000000000000.0100000000000000b =    0.25                 |
|  + 0000000000000001.1100000000000000b =  + 1.75                 |
|  ------------------------------------   --------                |
|    0000000000000010.0000000000000000b =    2.00                 |
+-----------------------------------------------------------------+
```

Realmente simples... é apenas uma soma binária... Suponha que tenhamos um númeroo em ponto fixo no registrador EAX e outro no EDX. O código para somar os dois números ficaria tão simples quanto:

```
+-----------------------------------------------------------------+
|  ADD     EAX,EDX                                                |
+-----------------------------------------------------------------+
```

O mesmo ocorre na subtração... Lógicamente, a subtração é uma adicão com o segundo operando complementado (complemento 2), então não há problemas em fazer:

```
+-----------------------------------------------------------------+
|  SUB     EAX,EDX                                                |
+-----------------------------------------------------------------+
```

A adição ou subtração de dois números em ponto fixo consome de 1 a 2 ciclos de máquina apenas, dependendo do processador... o mesmo não ocorre com aritimética em ponto-flutuante!

A complicação começa a surgir na multiplicação e divisão de dois números em ponto-fixo. Não podemos simplesmente multiplicar ou dividir como fazemos com a soma:

```
+-----------------------------------------------------------------+
|    0000000000000001.0000000000000000                            |
|  * 0000000000000001.0000000000000000                            |
| ------------------------------------                            |
|    0000000000000000.0000000000000000 + carry                    |
+-----------------------------------------------------------------+
```

Nultiplicando 1 por 1 deveriamos obter 1, e não 0. Vejamos a multiplicação de dois valores menores que 1 e maiores que 0:

```
+-----------------------------------------------------------------+
|    0000000000000000.1000000000000000     0.5                    |
|  * 0000000000000000.1000000000000000   * 0.5                    |
| ------------------------------------ -------                    |
|    0100000000000000.0000000000000000  16384.0                   |
+-----------------------------------------------------------------+
```

Hummm... o resultado deveria dar 0.25. Se dividirmos o resultado por 65536 (2^16) obteremos o resultado correto:

```
+-----------------------------------------------------------------+
|    0100000000000000.0000000000000000 >> 16 =                    |
|    0000000000000000.0100000000000000       = 0.25               |
+-----------------------------------------------------------------+
```

Ahhh... mas, e como ficam os números maiores ou iguais a 1?! A instrução IMUL dos microprocessadores 386 ou superiores permitem a multiplicação de dois inteiros de 32 bits resultando num inteiro de 64 bits (o resultado ficará em dois registradores de 32 bits separados!). Assim, para multiplicarmos dois números em ponto fixo estabelecemos a seguinte regra:

```
+-------------------------------------------------------------------+
|  resultado = (n1 * n2) / 65536          ou                        |
|  resultado = (n1 * n2) >> 16                                      |
+-------------------------------------------------------------------+
```

Assim, retornando ao primeiro caso de multiplicação (em notação hexa agora!):

```
+-------------------------------------------------------------------+
|  0001.0000h * 0001.0000h = 000000010000.0000h                     |
|                                                                   |
|  Efetuando o shift de 16 bits para a direita:                     |
|                                                                   |
|  00010000.0000h >> 16 = 0001.0000h                                |
|                                                                   |
+-------------------------------------------------------------------+
```

Em assembly isso seria tão simples como:

```
+-------------------------------------------------------------------+
|  PROC    FixedMul                                                 |
|  ARG     m1:DWORD, m2:DWORD                                       |
|                                                                   |
|      mov     eax,m1                                               |
|      mov     ebx,m2                                               |
|      imul    ebx                                                  |
|      shrd    eax,edx,16                                           |
|      ret                                                          |
|                                                                   |
|  ENDP                                                             |
+-------------------------------------------------------------------+
```

A instrução IMUL, e não MUL, foi usada porque os números de ponto fixo são sinalizados (o bit mais significativo é o sinal!). Vale aqui a mesma regra de sinalização para números inteiros: Se o bit mais significativo estiver setado o númeroo é negativo e seu valor absoluto é obtido através do seu complemento (complemento 2). Quanto a manipulação dos sinais numa multiplicação... deixe isso com o IMUL! :)

A divisão também tem as suas complicações... suponha a seguinte divisão:

```
+-------------------------------------------------------------------+
|   0001.0000h                                                      |
|  ------------ = 0000.0000h (resto = 0001.000h)                    |
|   0002.0000h                                                      |
+-------------------------------------------------------------------+
```

A explicação deste resultado é simples: estamos fazendo divisão de dois números inteiros... Na aritimética inteira a divisão com o dividendo menor que o divisor sempre resulta num quociente zero!

Eis a solução: Se o divisor está deslocado 16 bits para esquerda (20000h é diferente de 2, certo!?), então precisamos deslocar o dividendo 16 bits para esquerda antes de fazermos a divisão! Felizmente os processadores 386 e superiores permitem divisões com dividendos de 64bits e divisores de 32bits. Assim, o deslocamento de 16 bits para esquerda do dividendo não é problemática!

```
0001.0000h << 16 = 00010000.0000h

00010000.0000h / 0002.0000h = 0000.8000h

    ou seja:

1 / 2 = 0.5
```

Eis a rotina em assembly que demonstra esse algorritmo:

```
PROC    FixedDiv
ARG     d1:DWORD, d2:DWORD

    mov     eax,d1      ; pega dividendo
    mov     ebx,d2      ; pega divisor

    sub     edx,edx

    shld    edx,eax,16
    shl     eax,16

    idiv    ebx
    ret

ENDP
```

Isso tudo é muito interessante, não?! Hehehe... mas vou deixar vc mais desesperado ainda: A divisão tem um outro problema! E quanto aos sinais?! O bit mais significativo de um inteiro pode ser usado para sinalizar o númeroo (negativo = 1, positivo = 0), neste caso teremos ainda que complementar o númeroo para sabermos seu valor absoluto. Se simplesmente zeraramos EDX e o bit mais significativo estiver setado estaremos dividindo um númeroo positivo por outro númeroo qualquer (já que o bit mais significativo dos 64bits resultantes será 0!). Vamos complicar mais um pouquinho o código da divisão para sanar este problema:

```
PROC    FixedDiv
ARG     d1:DWORD, d2:DWORD

    sub     cl,cl       ; CL = flag
                        ; == 0 -> resultado positivo.
                        ; != 0 -> resultado negativo.

    mov     eax,d1      ; pega dividendo

    or      eax,eax     ; é negativo?!
    jns     @@no_chs1   ; não! então não troca sinal!

    neg     eax         ; é! então troca o sinal e...
    inc     cl          ; incrementa flag.
@@no_chs1:

    mov     ebx,d2      ; pega divisor

    or      ebx,ebx     ; é negativo?!
    jns     @@no_chs2   ; não! então não troca sinal!

    neg     ebx         ; é! então troca sinal e...
```

```
    dec     cl          ; decrementa flag.
@@no_chs2:

    sub     edx,edx

    shld    edx,eax,16
    shl     eax,16

    div     ebx         ; divisão de valores positivos...
                        ; ... não precisamos de idiv!

    or      cl,cl       ; flag == 0?
    jz      @@no_chs3   ; sim! resultado é positivo.

    neg     eax         ; não! resultado é negativo...
                        ; ... troca de sinal!
@@no_chs3:
    ret

ENDP
```

Se ambos os valores são negativos (d1 e d2) então o resultado será positivo. Note que se d1 é negativo CL é incrementado. Logo depois... se d2 também é negativo, CL é decrementado (retornando a 0). A rotina então efetuará divisão de valores positivos e somente no final é que mudará o sinal do resultado, se for necessário!

Uma consideração a fazer é: Como "transformo" um númeroo em ponto flutuante em ponto-fixo e vice-versa?!

Comecemos pela transformação de números inteiros em ponto-fixo: O nosso ponto-fixo está situado exatamente no meio de uma doubleword (DWORD), o que nos dá 16 bits de parte inteira e 16 de parte fracionária. A transformação de um númeroo inteiro para ponto-fixo é mais que simples:

```
FixP = I * 65536          ou
FixP = I << 16

onde FixP = Fixed Point (Ponto fixo)
     I    = Integer (Inteiro)
```

Desta forma os 16 bits superiores conterão o númeroo inteiro e os 16 bits inferiores estarão zerados (um inteiro não tem parte fracionária, tem?!).

Se quisermos obter a componente inteira de um númeroo de ponto fixo basta fazer o shift de 16 bits para direita.

A mesma regra pode ser usada para transformação de ponto-flutuante para ponto-fixo, só que não usaremos shifting e sim multiplicaremos explicitamente por 65536.0! Suponha que queiramos transforma o númeroo PI em ponto-fixo:

```
FixP = FloatP * 65536.0

FixP = 3.1415... * 65536.0 = 205887.4161
FixP = 205887

FixP = 0003.2439h
```

O que nos dá uma boa aproximação (se transformarmos 32439h em ponto flutuante novamente obteremos 3.14149475...). Apenas a parte inteira do resultado (205887.4161) nos interessa. (205887). Mas apareceu um pequenino problema que talvez vc não tenha notado...

Suponha que o resultado da multiplicação por 65536.0 desse 205887.865 (por exemplo, tá?!). Esse númeroo está mais próximo de 205888 do que de 205887! Se tomarmos apenas a componente inteira do resultado obteremos um erro ainda maior (ponto-fixo não é muito preciso, como vc pode notar pelo exemplo acima!). Como fazer para obter sempre a componente inteira mais aproximada?! A solução é somar 0.5 ao resultado da multiplicação por 65536.0!

Se a componente fracionária for maior ou igual a 0.5 então a soma da componente fracionária com 0.5 dará valor menor que 2.0 e maior ou igual a 1.0 (ou seja, a componente inteira dessa soma será sempre 1.0). Ao contrário, se a componente fracionária do resultado da multiplicação por 65536.0 for menor que 0.5 então a componente inteira da soma dessa componente por 0.5 será sempre 0.0! Então, somando o resultado da multiplicação com 0.5 podemos ou não incrementar a componente inteira de acordo com a proximidade do númeroo real com o inteiro mais próximo!

Se a aproximação não for feita, o erro gira em torno de 15e-6, ou seja: 0.000015 (erro a patir da quinta casa decimal!).

A transformação de um númeroo de ponto-flutuante para ponto-fixo fica então:

```
FixP = (FloatP * 65536.0) + 0.5

FixP = (3.1415... * 65536.0) + 0.5 = 205887.4161 + 0.5
FixP = 205887.9161
FixP = 205887  (ignorando a parte fracionária!)

FixP = 0003.2439h
```

A transformação contrária (de ponto-fixo para ponto-flutuante) é menos traumática, basta dividir o númeroo de ponto fixo por 65536.0. Eis algumas macros, em C, para as transformações:

```
#define INT2FIXED(x)    ((long)(x) << 16)
#define FIXED2INT(x)    ((x) >> 16)
#define DOUBLE2FIXED(x) (long)(((x) * 65536.0) + 0.5)
#define FIXED2DOUBLE(x) ((double)(x) / 65536.0)
```

Aritimética de ponto-fixo é recomendável apenas no caso de requerimento de velocidade e quando não necessitamos de precisão nos calculos. O menor númeroo que podemos armazenar na configuração atual é ±1.5259e-5 (1/65536) e o maior é ±32767.99998, aproximadamente. números maiores ou menores que esses não são representáveis. Se o seu programa pode extrapolar esta faixa, não use ponto-fixo, vc obterá muitos erros de precisão e, ocasionalmente, talvez até um erro de "Division By Zero".

Atenção... A implementação dos procedimentos (PROC) acima são um pouquinho diferentes para mixagem de código... Os compiladores C e PASCAL atuais utilizam o par DX:AX para retornar um DWORD, assim, no fim de cada PROC e antes do retorno coloque:

```
  shld    edx,eax,16
  shr     eax,16
```

Ou faça melhor ainda: modifique os códigos!

Eis a minha implementação para as rotinas FixedMul e FixedDiv para mixagem de código com C ou TURBO PASCAL:

```
/*
** Arquivo de cabeçalho FIXED.H
*/
#if !defined(__FIXED_H__)
#define __FIXED_T__

/* Tipagem */
typedef long    fixed_t;

/* Macros de conversão */
#define INT2FIXED(x)    ((fixed_t)(x) << 16)
#define FIXED2INT(x)    ((int)((x) >> 16))
#define DOUBLE2FIXED(x) ((fixed_t)(((x) * 65536.0) + 0.5))
#define FIXED2DOUBLE(x) ((double)(x) / 65536.0)

/* Declaração das funções */
fixed_t pascal FixedMul(fixed_t, fixed_t);
fixed_t pascal FixedDiv(fixed_t, fixed_t);

#endif
```

```
{*** Unit FixedPt para TURBO PASCAL ***}
UNIT FIXEDPT;

{} INTERFACE {}

{*** Tipagem ***}
TYPE
    TFixed  = LongInt;

{*** Declaração das funções ***}
FUNCTION FixedMul(M1, M2 : TFixed) : TFixed;
FUNCTION FixedDiv(D1, D2 : TFixed) : TFixed;

{} IMPLEMENTATION {}

{*** Inclui o arquivo .OBJ compilado do código abaixo ***}
{$L FIXED.OBJ}

{*** Declara funções como externas ***}
FUNCTION FixedMul(M1, M2 : TFixed) : TFixed; EXTERN;
FUNCTION FixedDiv(D1, D2 : TFixed) : TFixed; EXTERN;

{*** Fim da Unit... sem inicializações! ***}
END.
```

```
; FIXED.ASM
; Módulo ASM das rotinas de multiplicação e divisão em
; ponto fixo.

; Modelamento de memória e modo do compilador.
IDEAL
MODEL LARGE,PASCAL
LOCALS
JUMPS
P386        ; Habilita instruções do 386

; Declara os procedimentos como públicos
GLOBAL FixedMul : PROC
GLOBAL FixedDiv : PROC

; Inicio do segmento de código.
CODESEG

PROC    FixedMul
ARG     m1:DWORD, m2:DWORD

    mov     eax,[m1]
    mov     ebx,[m2]
    imul    ebx
    shr     eax,16  ; Coloca parte fracionária em AX.
                    ; DX já contém parte inteira!
    ret

ENDP

; Divisão em ponto fixo.
; d1 = Dividendo, d2 = Divisor
PROC    FixedDiv
ARG     d1:DWORD, d2:DWORD

    sub     cl,cl       ; CL = flag
                        ; == 0 -> resultado positivo.
                        ; != 0 -> resultado negativo.

    mov     eax,[d1]    ; pega dividendo

    or      eax,eax     ; é negativo?!
    jns     @@no_chs1   ; não! então não troca sinal!

    neg     eax         ; é! então troca o sinal e...
    inc     cl          ; incrementa flag.
@@no_chs1:

    mov     ebx,[d2]    ; pega divisor

    or      ebx,ebx     ; é negativo?!
    jns     @@no_chs2   ; não! então não troca sinal!

    neg     ebx         ; é! então troca sinal e...
    dec     cl          ; decrementa flag.
@@no_chs2:

    sub     edx,edx     ; Prepara para divisão.
    shld    edx,eax,16
    shl     eax,16

    div     ebx         ; divisão de valores positivos...
```

```
                        ; ... não precisamos de idiv!

    or      cl,cl       ; flag == 0?
    jz      @@no_chs3   ; sim! resultado é positivo.

    neg     eax         ; não! resultado é negativo...
                        ; ... troca de sinal!
@@no_chs3:

    ;
    ; Apenas adequa para o compilador
    ;
    shld    edx,eax,16  ; DX:AX contém o DWORD
    shr     eax,16

    ret

ENDP

END
```

+--------------+
¦ ASSEMBLY XXI ¦
+--------------+

Olá!!... Acho que você concorda comigo que essa série de textos não estaria completa se eu não falasse alguma coisa a respeito de programação da placa de vídeo VGA, né?! Acho que nós temos razão em pensar assim! :)

Inicialmente começarei a descrever a placa VGA, depois vem as descrições da SVGA e VESA. Não pretendo gastar "trocentas" horas de digitação e depuração de código na descrição desses padrões.. quero apenas dar uma idéia geral do funcionamento desses dispositivos para que você possa caminhar com as próprias pernas mais tarde...

¦ Video Graphics Array

O padrão VGA é o sucessor dos padrões EGA e CGA, todos criados pela IBM... A diferença básica do VGA para os outros dois é o aumento da resolução e de cores. Eis uma comparação dos modos de maior resolução e cores desses três padrões (aqui estão listados apenas os modos gráficos!):

	CGA	EGA	VGA
Maior resolução	640x200	640x350	640x480
Maior númeroo de cores	4 (320x200)	16 (640x350)	16 (640x480) 256 (320x200)

O padrão VGA suporta até 256 cores simultanemente no modo de vídeo 13h (320x200x256). E no modo de mais alta resolução suporta o mesmo númeroo de cores que a EGA, que são apenas 16.

Quanto ao númeroo de cores, as placas EGA e VGA são mais flexíveis que a irmã mais velha (a CGA). As cores são "reprogramáveis", isto é, de uma palette de 256k cores (256 * 1024 = 262144 cores), na VGA, podemos escolher 256... Duma palette de 64 cores podemos usar 16, na EGA... A VGA é, sem sombra de dúvidas, superior!

A forma como podemos selecionar essas cores todas será mostrada mais abaixo (Como sempre as coisas boas são sempre deixadas pra depois, né?! hehe).

Em tempo: O modo 640x480 (16 cores) será usado como exemplo nas próximas listagens dos textos daqui pra frente... O modo gráfico de 320x200 com 256 cores será discutido em outra oportunidade, bem como o famoso MODE X (modo de vídeo não documentado da VGA - e largamente descrito por Michael Abrash em seus artigos para a revista Dr. Dobb's).

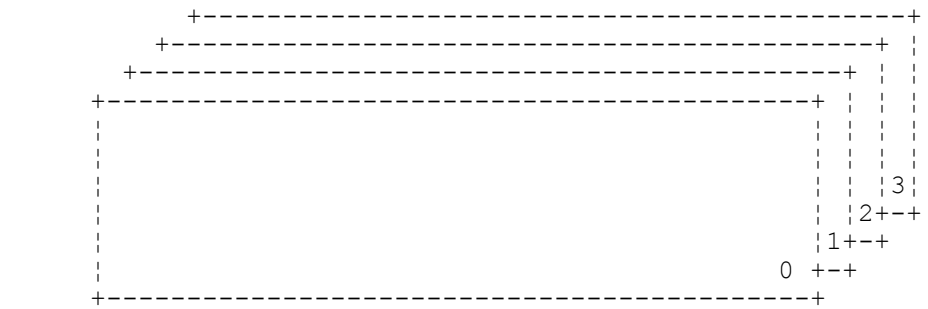
¦ Memória de vídeo

Existe um grande obstáculo com relação a modos gráficos de resoluções altas: A segmentação de memória! Lembre-se que os processadores Intel enxergam a memória como blocos de 64k não

sequenciados (na verdade, sobrepostos!)... No modo gráfico de resolução 640x480 da VGA (que suporta 16 cores no máximo), suponha que cada byte da memória de vídeo armazenasse 2 pixeis (16 cores poderia equivaler a 4 bits, não poderia?!)... Well isso nos dá 320 bytes por linha (meio byte por pixel -> 640 / 2 = 320!).

Com os 320 bytes por linha e 480 linhas teriamos 153600 bytes numa tela cheia! Ocupando 3 segmentos da memória de vídeo (2 segmentos contíguos completos e mais 22528 bytes do terceiro!)... Puts... Imagine a complexidade do algoritmo que escreve apenas um ponto no vídeo! Seria necessário selecionarmos o segmento do pixel e o offset... isso pra aplicativos gráficos de alta performance seria um desastre!

A IBM resolveu esse tipo de problema criando "planos" de memória... Cada plano equivale a um bit de um pixel. Dessa forma, se em um byte temos oito bits e cada plano armazena 1 bit de 1 pixel... em um byte de cada plano teremos os 8 bits de 8 pixeis. Algo como: O byte no plano 0 tem os oito bits 0 de oito pixeis... no plano 1 temos os oito bits 1 de oito pixeis... e assim por diante. De forma que o circuito da VGA possa "sobrepor" os planos para formar os quatro bits de um único pixel... A representação gráfica abaixo mostra a sobreposição dos planos:



Esses são os quatro planos da memória de vídeo. O plano da frente é o plano 0, incrementando nos planos mais interiores. Suponha que na posição inicial de cada plano tenhamos os sequintes bytes:

```
+--------------------------------------------------------------------+
¦  Plano 0: 00101001b                                                ¦
¦  Plano 1: 10101101b                                                ¦
¦  Plano 2: 11010111b                                                ¦
¦  Plano 3: 01010100b                                                ¦
+--------------------------------------------------------------------+
```

Os bits mais significativos de cada plano formam um pixel: (0110b), os bits seguintes o segundo pixel (0011b), o terceiro (1100b), e assim por diante até o oitavo pixel (1110b). Como temos 16 cores no modo 640x480, cada pixel tem 4 bits de tamanho.

Com esse esquema biruta temos um espaço de apenas 38400 bytes sendo usados para cada plano de vídeo... Se cada byte suporta um bit de cada pixel então temos que uma linha tem 80 bytes de tamanho (640 / 8). Se temos 480 linhas, teremos 38400 bytes por plano.

Tome nota de duas coisas... estamos usando um modo de 16 cores como exemplo para facilitar o entendimento (os modos de 256 cores são mais complexos!) e esses 38400 bytes em cada plano de bits é um espaço de memória que pertence à placa de vídeo e é INACESSíVEL a CPU!!!! Apenas a placa de vídeo pode ler e gravar nessa memória. A placa VGA (e também a EGA) usam a memória RAM do sistema para

saberem quais posições de um (ou mais) planos de bits serão afetados. Isso é assunto para o próximo tópico:

¦ A memória do sistema:

Os adaptadores VGA usam o espaço de "memória linear" entre 0A0000h e 0BFFFFh (todo o segmento 0A000h e todo o segmento 0B000h)... Essa memória é apenas uma área de rascunho, já que a placa VGA tem memória própria... A CPU precisa de uma memória fisicamente presente para que possa escrever/ler dados... daí a existencia desses dois segmentos contíguos de memória, mas a VGA não os usa da mesma forma que a CPU!

Citei dois segmentos contíguos... mas não existe a limitação de apenas um segmento?! Well... existe... o segmento 0B000h é usado apenas nos modos-texto (onde o segmento 0B800h é usado... 0B000h é para o adaptador monocromático - MDA)... os modos-gráficos utilizam o segmento 0A000h (a não ser aqueles modos gráficos compatíveis com a CGA!).

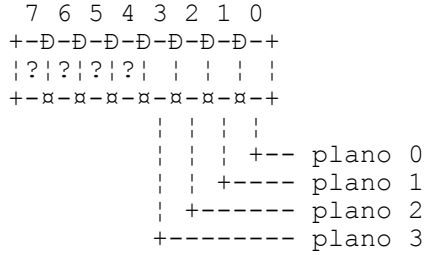
A memória do sistema é usada como rascunho pela VGA (e pela EGA também!!)... A VGA colhe as modificações feitas na memória do sistema e transfere para a memória de vídeo. A forma com que isso é feito depende do modo com que programamos a placa de vídeo para fazê-lo... podemos modificar um plano de bits por vez ou vários planos, um bit por vez, vários bits de uma vez, etc. Na realidade, dependendo do modo com que os dados são enviados para a placa VGA não precisamos nem ao menos saber O QUE estamos escrevendo na memória do sistema, a VGA toma conta de ajustar a memória de vídeo por si só, usando apenas o endereço fornecido pela CPU para saber ONDE deve fazer a modificação!

¦ Selecionando os planos de bits...

Em todos os modos de escrita precisamos selecionar os planos de bits que serão afetados... Isso é feito através de um registrador da placa VGA: MapMask... Porém, antes de sairmos futucando tudo quanto é endereço de I/O da placa VGA precisamos saber COMO devemos usá-los!

A maioria dos registradores da placa VGA estão disponíveis da seguinte maneira: Primeiro informamos à placa qual é o registrador que queremos acessar e depois informamos o dado a ser escrito ou lido... A técnica é a seguinte: escrevemos num endereço de I/O o númeroo do registrador... no endereço seguinte o dado pode ser lido ou escrito...

No caso de MapMask, este registrador é o númeroo 2 do CIRCUITO SEQUENCIADOR da placa VGA. O circuito sequenciador pode ser acessado pelos endereços de I/O 3C4h e 3C5h (3C4h conterá o númeroo do registro e 3C5h o dado!). Eis a estrutura do registro MapMask:



De acordo com o desenho acima... os quatro bits inferiores informam a placa VGA qual dos planos será modificado. Lembre-se que cada plano tem um bit de um pixel (sendo o plano 0 o proprietário do bit menos significativo). Vamos a nossa primeira rotina:

```
; VGA1.ASM
; Compile com:
;
;   TASM vga1
;   TLINK /x/t vga1
;
ideal
model tiny
locals
jumps

codeseg

org 100h
start:
    mov     ax,12h      ; Poe no modo 640x480
    int     10h

    mov     ax,0A000h   ; Faz ES = 0A000h
    mov     es,ax
    sub     bx,bx       ; BX será o offset!

    mov     dx,03C4h    ; Aponta para o registro
    mov     al,2        ; "MapMask"
    out     dx,al

    inc     dx          ; Incrementa endereço de I/O

    mov     al,0001b    ; Ajusta para o plano 0
    out     dx,al

    mov     [byte es:bx],0FFh   ; Escreve 0FFh

    mov     al,0100b    ; Ajusta para o plano 2
    out     dx,al

    mov     [byte es:bx],0FFh   ; Escreve 0FFh

    sub     ah,ah       ; Espera uma tecla!
    int     16h         ; ... senão não tem graça!!! :)

    mov     ax,3        ; Volta p/ modo texto 80x25
    int     10h

    int     20h         ; Fim do prog

end start
```

Depois de compilar e rodar o VGA1.COM você vai ver uma pequena linha magenta no canto superior esquerdo do vídeo... Se você quiser que apenas o pixel em (0,0) seja aceso, então mude o valor 0FFh nas instruções "mov [byte es:bx],0FFh" para 80h. O motivo para isso é que cada byte tem apenas um bit de um pixel, isto é, cada bit do byte equivale a um bit do pixel... necessitamos alterar os quatro planos de bits para setarmos os quatro bits de cada pixel (quatro bits nos dão 16 combinações)... assim, se um byte tem oito bits, o primeiro byte dos quatro planos de bits tem os oito pixeis iniciais,

sendo o bit mais significativo do primeiro byte de cada plano o primeiro pixel.

Deu pra notar que apenas modificamos os planos 0 e 2, né?! Notamos também que desta maneira não temos como alterarar um único pixel... sempre alteraremos os oito pixels!! Mas, não se preocupe... existem outros recursos na placa VGA... Entendendo o esquema de "planos de bits" já está bom por enquando...

Até a próxima...

+---------------+
¦ ASSEMBLY XXII ¦
+---------------+

Alguma vez aconteceu de você ter aquela rotina quase concluída e quando foi testá-la viu que estava faltando alguma coisa?! Bem... se não aconteceu você é um sortudo... Quando eu estava começando a entender o funcionamento da placa VGA me dispus a construir rotinas básicas de traçagem de linhas horizontais e verticais... porém, quando tinha algum bitmap atrás da linha acontecia uma desgraça!!! Parte do bitmap sumia ou era substituído por uma sujeirinha chata!

Obviamente eu ainda não tinha dominado o funcionamento da placa... por isso, vamos continuar com os nossos estudos...

¦ A mascara de bits e os LATCHES da VGA.

Existe uma maneira de não alterarmos bits indesejáveis em um byte de cada plano... Suponha que queiramos modificar apenas o bit mais significativo de um byte nos planos de bits, deixando o restante exatamente como estavam antes!

Well... Isso pode ser feito de duas formas: Primeiro lemos o byte de um plano, realizamos um OR ou um AND com esse byte e o byte com o bit a ser alterado (zerando-o ou setando-o de acordo com a modificação que faremos... veja as instruções AND e OR num dos textos iniciais do curso de ASM para ter um exemplo de como isso pode ser feito!)... depois da operação lógica, escrevemos o byte na mesma posição... Essa é a maneira mais dispendiosa!

A placa VGA permite que criemos uma mascara de bits para podermos alterar apenas aqueles bits desejados... Isso é feito pelo registrador BitMask. Mas, antes temos que ler o byte inteiro... hummm... acontece que existe um registrador intermediário, interno, que retém o último byte lido de um plano de bits... esse registrador é conhecido como LATCH.

Basta ler um byte da memória do sistema que os bytes dos quatro planos de bits vão para seus LATCHES... Depois precisamos mascarar os bits que não queremos modificar no registrador BitMask para só então escrever na memória do sistema (no plano de bits!)... Não esquecendo de setar os planos de bits que queremos alterar via MapMask, como visto no último texto!

O funcionamento dos latches em conjunto com BitMask é o seguinte: Uma vez carregados os latches, apenas os bits ZERADOS de BitMask serão copiados de volta para os planos de bits selecionados por MapMask. Em contrapartida, os bits SETADOS em BitMask correspondem aos bits vindos da memória do sistema, que são fornecidos pela CPU. Dessa maneira a nossa rotina não tem que propriamente ler o conteúdo de um plano de bits (aliás, o que for lido pela CPU pode muito bem ser ignorado!)... não necessitamos nem ao menos efetuar operações lógicas para setar ou resetar um determinado bit do byte que será escrito num plano de bits!

Vímos no último texto que o registro MapMask faz parte do circuito SEQUENCIADOR da VGA. O registro BitMask está localizado em outro circuito. Mais exatamente no controlador gráfico (Graphics Controller - que chamaremos de GC)... O funcionamento é o mesmo do que o circuito sequenciador, em termos de endereços de I/O, citado no último texto: Primeiro devemos informar o númeroo do registro e depois o valor. O GC pode ser acessado a partir do endereço de I/O 03CEh e o númeroo do registro BitMask é 8.

Eis nosso segundo exemplo:

```
; VGA2.ASM
; Compile com:
;
;   TASM vga2
;   TLINK /x/t vga2
;
ideal
model tiny
locals
jumps

codeseg

org 100h
start:
    mov     ax,12h      ; Poe no modo 640x480
    int     10h

    mov     ax,0A000h   ; Faz ES = 0A000h
    mov     es,ax
    sub     bx,bx       ; BX será o offset!

    mov     dx,03C4h    ; Seleciona planos 0 e 2...

    mov     ax,0502h    ; ídem a fazer: mov al,2
                        ;               mov ah,0101b

    out     dx,ax

    mov     dx,03CEh    ; Mascara todos os bits,
    mov     ax,8008h    ;  exceto o bit 7
    out     dx,ax

    mov     al,[byte es:bx]     ; carrega os latches da VGA
                                ;  note que AL não nos
                                ;  interessa!!!
    mov     [byte es:bx],0FFh   ; Escreve 0FFh

    sub     ah,ah       ; Espera uma tecla!
    int     16h         ; ... senão não tem graça!!! :)

    mov     ax,3        ; Volta p/ modo texto 80x25
    int     10h

    int     20h         ; Fim do prog

end start
```

Temos algumas novidades aqui... Primeiro: é possível escrever o númeroo de um registro e o dado quase que ao mesmo tempo... basta usar a instrunção OUT DX,AX - recorra a textos anteriores para ver o funcionamento dessa instrução!. Segundo: mesmo escrevendo 0FFh (todos os bits setados) na memória do sistema, apenas o bit que não está mascarado será modificado, graças ao BitMask!! Terceiro: Mais de um plano de bits pode ser alterado ao mesmo tempo! Note que nesse código escrevemos na memória de vídeo apenas uma vez e os planos 0 e 2 foram alterados (continua a cor MAGENTA, não?!).

¦ Problemas à vista!

Ok... aparentemente a coisa funciona bem... dai eu faço uma simples pergunta: O que aconteceria se o ponto em (0,0) estivesse inicialmente "branco" e usassemos a rotina acima?!

Hummmm... Se o ponto é branco, a cor é 15... 15 é 1111b em binário, ou seja, todos os planos de bits teriam o bit 7 do primeiro byte setados... A rotina acima "seta" os bits 7 do primeiro byte dos planos 0 e 2... assim a cor CONTINUARIA branca!! MAS COMO SOU TEIMOSO, EU QUERO MAGENTA!!!

A solução seria colocar as seguintes linhas antes da instrução "sub ah,ah" na listagem acima:

```
¦  mov     dx,03C4h    ; Seleciona os planos 1 e 3                ¦
¦  mov     ax,0A02h                                               ¦
¦  out     dx,ax                                                  ¦
¦                                                                 ¦
¦  mov     [byte es:bx],0 ; escreve 0 nos planos 1 e 3            ¦
```

Precisamos zerar os bits 7 dos planos 1 e 3... Note que nas linhas acima não carreguei os latches da VGA através de leitura... aliás... não carreguei de forma alguma. Não preciso fazer isso os latches dos planos 1 e 3 não foram alterados desde a sua última leitura... repare que não "desmascarei" os bits no registro BitMask... dai não ter a necessidade de mascará-los de novo... só preciso escrever 0 nos planos 1 e 3 para que o bit 7 seja alterado.

Puts... que mão-de-obra!!... Felizmente existem meios mais simples de fazer isso tudo... Ahhhhhh, mas é claro que isso fica pra um próximo texto! :))

+----------------+
¦ ASSEMBLY XXIII ¦
+----------------+

Confesso a todos vocês que a experiência que venho tendo com relação a programação da placa VGA começou com a leitura de artigos e de um livro de um camarada chamado Michael Abrash... Gostaria muito de conseguir outros livros desse sujeito!! Aliás, se puderem colocar as mãos num livrão chamado "Zen of Graphics Programming", garanto que não haverá arrependimentos! É um excelente livro com MUITOS macetes, rotinas e explicações sobre a VGA... Tudo isso com bom humor!!! :)

Outra boa aquisição, pelo menos com relação ao capítulo 10, é o livro "Guia do Programador para as placas EGA e VGA" da editora CIENCIA MODERNA (o autor é Richard E. Ferraro). Explicitei o capítulo 10 porque acho que esse livro só não é tão bom devido a falhas de tradução (coisa que acontece com quase todos os livros traduzidos no Brasil!)... O capítulo 10 é tão somente uma referência (enorme e confusa, mas quebra bem o galho) a todos os registradores da VGA. Esse é um dos livros que adoraria poder ter o original, em inglês!

¦ Onde paramos?!

Ahhh... sim... até aqui vimos o modo de escrita "normal" da placa VGA. Esse modo de escrita é o usado pela BIOS e é conhecido como "modo de escrita 0". Antes de passarmos pra outros modos de escrita vale a pena ver o funcionamento de outros dois registradores: o "Enable Set/Reset" e o "Set/Reset". Esses registros, como você vai ver, facilita muito o trabalho de escrita nos planos de bits.

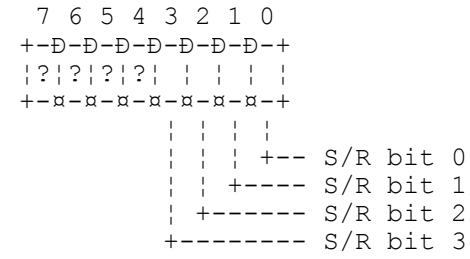
¦ Ligando e desligando bits...

Na listagem do text 22 vimos que é possível a escrita em mais de um plano de bits ao mesmo tempo (basta habilitar em MapMask). Vimos também que os planos de bits não habilitados para escrita via MapMask não são automaticamente zerados... lembra-se do caso do pixel branco que queriamos transformar em magenta?!

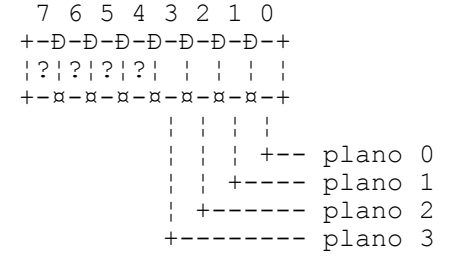
Com tudo isso, tinhamos que fazer pelo menos 3 acessos à memória do sistema: Uma leitura para carregar os latches, uma escrita para setar bits nos planos selecionados, e mais uma escrita para zerar os bits dos outros planos... Isso sem contar com os registradores que teremos que atualizar: MapMask e BitMask. Surpreendentemente a instrução OUT é uma das que mais consomem ciclos de máquina da CPU (especialmente nos 386s e 486s! Veja no seu HELP_PC).

Na tentativa de reduzir os acessos à memória do sistema (e indiretamenta aos planos de bits!), lançaremos mão dos registradores "Enable Set/Reset" e "Set/Reset". Eis a descrição deles:

» REGISTRO ENABLE SET/RESET



» REGISTRO SET/RESET



O registrador "Enable Set/Reset" informa a placa VGA quais bits do registrador "Set/Reset" vão ser transferidos para os planos de bits. Note que cada bit de "Set/Reset" está associado a um plano de bits! Os bits não habilitados em "Enable Set/Reset" virão da CPU ou dos latches, dependendo do conteúdo de BitMask - como vimos no exemplo do texto 22.

Não sei se você percebeu, mas podemos agora escrever quatro bits diferentes nos quatro planos de bits ao mesmo tempo... Se setarmos os quatro bits de "Enable Set/Reset", os quatro bits em "Set/Reset" serão transferidos para a memória de vídeo. Nesse caso o que a CPU enviar para a memória do sistema será ignorado (já que é "Set/Reset" que está fornecendo os dados!).

Os registradores MapMask e BitMask continuam funcionando como antes... Se não habilitarmos um ou mais planos de bits em MapMask, este(s) plano(s) não será(ão) atualizado(s)! Note que "Enable Set/Reset" diz ao circuito da placa VGA que deve ler os respectivos bits de "Set/Reset" e colocá-los nos respectivos planos de bits... mas, MapMask pode ou não permitir essa transferência!!! Quanto ao registrador BitMask, vai bem obrigado (veja discussão sobre ele no texto anterior).

Hummm... virou bagunça! Agora podemos ter dados vindos de três fontes: da CPU (via memória do sistema), dos latches, e do registrador Set/Reset. Bem... podemos até usar essa bagunça em nosso favor!

"Enable Set/Reset" e "Set/Reset" pertencem ao mesmo circuito de BitMask: o controlador gráfico (GC). Só que o índice (que é o número do registro no circuito!) de "Set/Reset" é 0 e de "Enable Set/Reset" é 1.

Vamos a um exemplo com esses dois registradores:

```
; VGA3.ASM
; Compile com:
;
;   TASM vga3
;   TLINK /x/t vga3
;
ideal
model tiny
locals
jumps

codeseg

org 100h
start:
    mov     ax,12h      ; Poe no modo 640x480
    int     10h

    mov     ax,0A000h   ; Faz ES = 0A000h
    mov     es,ax
    sub     bx,bx       ; BX será o offset!

    mov     dx,03C4h
    mov     ax,0F02h    ; MapMask = 1111b
    out     dx,ax

    mov     dx,03CEh
    mov     ax,8008h    ; BitMask = 10000000b
    out     dx,ax
    mov     ax,0500h    ; Set/Reset = 0101b
    out     dx,ax
    mov     ax,0F01h    ; Enable Set/Reset = 1111b
    out     dx,ax

    mov     al,[byte es:bx]     ; carrega os latches da VGA
                                ;  note que AL não nos
                                ;  interessa!!!
                                ; Isso é necessário pq vamos
                                ;  alterar apenas o bit 7. Os
                                ;  demais são fornecidos pelos
                                ;  latches.

    mov     [byte es:bx],al     ; Escreve qualquer coisa...
                                ;  AL aqui também não nos
                                ;  interessa, já que Set/Reset
                                ;  é quem manda os dados para
                                ;  os planos de bits.

    sub     ah,ah       ; Espera uma tecla!
    int     16h         ; ... senão não tem graça!!! :)

    mov     ax,3        ; Volta p/ modo texto 80x25
    int     10h

    int     20h         ; Fim do prog

end start
```

Explicando a listagem acima: Os quatro planos são habilitados em MapMask... depois habilitamos somente o bit 7 em BitMask, seguido

pela habilitação dos quatro bits de "Set/Reset" em "Enable Set/Reset". Uma vez que os quatro planos estão habilitados (por MapMask) e que os quatro bits de "Set/Reset" também estão (via "Enable Set/Reset"), colocamos em "Set/Reset" os quatro bits que queremos que sejam escritos nos planos: 0101b (ou 05h). Pois bem... precisamos apenas carregar os latches e depois escrever na memória do sistema.

Tudo bem, vc diz, mas qual é a grande vantagem?! Ora, ora... temos condições de alterar os quatro planos de bits ao mesmo tempo!! E, melhor ainda, estamos em condição de setar até oito pixeis ao mesmo tempo!!!! Experimente trocar a linha:

```
+-------------------------------------------------------------------+
¦      mov     ax,8008h    ; BitMask = 10000000b                    ¦
+-------------------------------------------------------------------+
```

por:

```
+-------------------------------------------------------------------+
¦      mov     ax,0FF08h   ; BitMask = 11111111b                    ¦
+-------------------------------------------------------------------+
```

Você verá oito pixeis magenta com uma única escrita na memória do sistema!!

Outra grande vantagem é o ganho de velocidade: Na listagem acima os dados que vão ser colocados nos planos de bits não são fornecidos diretamente pela CPU, mas sim por "Set/Reset" e pelos latches. Assim, a placa VGA não se interessa pelo conteúdo de AL que foi escrito na memória do sistema e não adiciona WAIT STATES, já que esse dado não vai para a memória de vídeo (fica só na memória do sistema!!).

É um grande avanço, né?! Well... próximos avanços nos próximos textos.

+---------------+
¦ ASSEMBLY XXIV ¦
+---------------+

Até agora vimos os registradores MapMask, BitMask, "Enable Set/Reset" e Set/Reset. Vimos também que MapMask permite ou não mudanças nos quatro planos de bits idependentemente. BitMask mascara os bits não desejáveis (e esses são lidos dos latches quando escrevemos na memória). Ainda por cima, vimos que é possível atualizar os quatro planos de bits ao mesmo tempo com bits diferentes usando "Enable Set/Reset" e Set/Reset. Isso tudo usando o modo de escrita 0!

¦ Modo de escrita 1

O modo de escrita 1 lida somente com os latches da placa VGA. Com esse modo podemos copiar o conteúdo dos quatro planos de bits de uma posição para outra com uma única instrução em assembly!

Como já vimos, os latches dos quatro planos são carregados sempre que fazemos uma leitura na memória do sistema (em todos os modos de escrita!). No modo 1 isso também vale. Só que nesse modo não é possível escrever nada nos planos de bits!! Simplesmente, quanto mandamos escrever numa determinada posição da memória do sistema, os latches é que atualizarão essa posição. No modo 1 os registros Set/Reset, "Enable Set/Reset" e BitMask não funcionam para nada. Assim, depois de setado o modo 1, podemos usar:

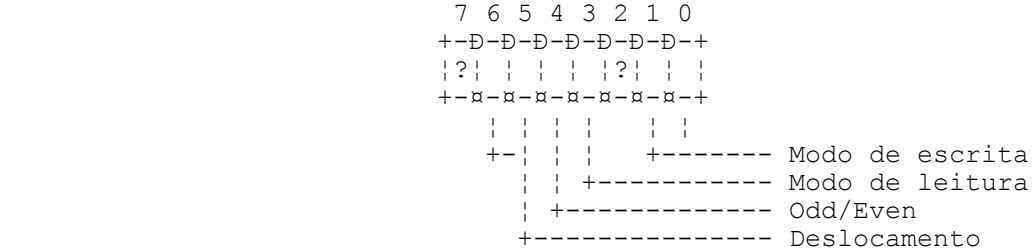
```
+-----------------------------------------------------------------+
¦  REP MOVSB                                                      ¦
+-----------------------------------------------------------------+
```

Para copiarmos bytes dos quatro planos de vídeo de uma posição da tela para outra. E RAPIDO! Só que tem um pequeno problema: Podemos copiar BYTES e não pixeis individuais! Lembre-se que um byte contém oito pixeis (com cada bit de um pixel em um plano de bits!). Se sua intenção é copiar um bloco inteiro, porém alinhado por BYTE, então o modo 1 é a escolha mais sensata. Caso contrário, use outro modo de escrita (o modo 0, por exemplo!).

Ahhh... podemos conseguir o mesmo efeito do modo de escrita 1 no modo de escrita 0! Basta zerarmos todos os bits de BitMask! Pense bem: Se BitMask está completamente zerado, então os dados virão apenas dos latches! O que nos deixa com um modo de escrita obsoleto, já que podemos fazer o mesmo trabalho no modo 0! :)

¦ O registrador MODE

Para ajustar o modo de escrita precisamos de um registrador. O registrador MODE é descrito abaixo:



O único campo que nos interessa no momento é o "Modo de escrita". Por isso, para modificar o modo, precisaremos ler o registro MODE, setar o modo de escrita, e depois reescrevê-lo... para que não façamos mudanças nos demais bits. Os modos de escrita válidos são os citados anteriormente (repare que esse campo tem 2 bits de tamanho!).

O registrador MODE faz parte do circuito GC (o mesmo de BitMask, "Enable Set/Reset" e Set/Reset) da placa VGA, seu índice é 5.

Well... já que o modo 1 é obsoleto, vou colocar aqui alguns macros para facilitar o entendimento dos próximos códigos-fonte, ok?

```
; VGA.INC
; Macros para VGA!
; Todos os macros alteram dx e ax

; Macro: Ajusta o modo de escrita
macro   SetWriteMode    mode
        ifdifi <mode>,<ah>
            mov     ah,mode
        endif
        mov     dx,3CEh
        mov     al,5
        out     dx,al
        inc     dx
        in      al,dx
        and     ax,1111111100b
        or      al,ah
        out     dx,al
endm

; Macro: Habilita/Mascara os planos de vídeo
macro   MapMask plane
        ifdifi  <plane>,<ah>
            mov     ah,plane
        endif
        mov     al,2
        mov     dx,3C4h
        out     dx,ax
endm

; Macro: Habilita os bits
macro   BitMask bit
        ifdifi  <bit>,<ah>
            mov     ah,bit
        endif
        mov     al,8
        mov     dx,3CEh
        out     dx,ax
endm

; Macro: Altera "Enable Set/Reset"
macro EnableSetReset    bitmsk
        ifdifi  <bitmsk>,<ah>
            mov     ah,bitmsk
        endif
        mov     al,1
        mov     dx,3CEh
        out     dx,ax
endm
```

```
; Macro: Ajusta Set/Reset
macro SetReset value
        ifdifi  <value>,<ah>
            mov     ah,value
        endif
        sub     al,al       ; altera tb os flags..
        mov     dx,3CEh
        out     dx,ax
endm
```

O modo de escrita 1 não é tão útil, como vimos no último texto... A plca VGA possui algumas redundancias que podem parecer desnessesárias à primeira vista, como por exemplo o modo de escrita 3. Nesse modo podemos despresar o registrador "Enable Set/Reset" e usar "Set/Reset" para ajustar os bits dos quatro planos de vídeo.

¦ Modo de escrita 3

Well... No modo 0 vimos como atualizar os quatro planos de bits de uma só vez... Isso é feito setando o registrador "Enable Set/Reset" e "Set/Reset"... usando também MapMask e BitMask para habilitarmos os planos e os bits desejados, respectivamente. Acontece que no modo 0 podemos ter uma mistura de dados vindos da CPU, dos latches e do registro Set/Reset... a mistura pode ser tão confusa que podemos ter a CPU atualizando um plano e Set/Reset outro. É, sem sombra de dúvida, um recurso interessante e bastante útil... mas se não tomarmos cuidado pode ser uma catastrofe, em termos visuais!

O modo de escrita 3 trabalha da mesma forma que o modo 0 só que "seta" automaticamente os quatro bits de "Enable Set/Reset". Isto é, a CPU não escreve nada nos planos de bits... isso fica sob responsabilidade do registrador "Set/Reset". O que a CPU escreve na memória so sistema sofre uma operação lógica AND com o conteúdo atual de BitMask... O resultado é usado como se fosse o BitMask! (Para facilitar as coisas... se BitMask for 11111111b e a CPU escrever 01100011b, então o "novo" BitMask será 01100011b, sem que o registrador BitMask seja afetado!!)

Com esse modo de escrita descartamos a necessidade de ajustar "Enable Set/Reset", eliminando a confusão que pode ser causada no modo 0... descartamos a atualização de BitMask, que pode feita indiretamente pela CPU... Mas, infelizmente não descartamos a necessidade de leitura da memória do sistema para carga dos latches e nem mesmo a necessidade de habilitarmos os planos de bits em MapMask! Se MapMask estiver zerado nenhum plano de bit será atualizado, lembre-se sempre disso!!! Isso é válido para TODOS os modos de escrita!

Eis um exemplo prático do uso do modo de escrita 3... Uma rotina que traça uma linha horizontal:

```
¦  ideal
¦  model small,c
¦  locals
¦  jumps
¦  p386
¦
¦  ; inclui os macros definidos no último texto!
¦  include "VGA.INC"
¦
¦  SCREEN_SEGMENT  equ 0A000h
¦
¦  ; Tamanho de uma linha... (modo 640x480)
¦  LINE_SIZE       equ 80
¦
¦  ; Coordenadas máximas...
¦  MAX_X_POS       equ 639
```

```
MAX_Y_POS       equ 479

global  grHorizLine:proc
global  grVertLine:proc
global  setGraphMode:proc
global  setTextMode:proc

codeseg

;*** DESENHA LINHA HORIZONTAL ***
proc    grHorizLine
arg     left:word, right:word, y:word, color:word
local   bitmask1:byte, bitmask2:byte
uses    si, di

        ; Verifica se a coordenada Y é válida...
        mov     ax,[y]
        or      ax,ax
        js      @@grHorizLineExit

        cmp     ax,MAX_Y_POS
        ja      @@grHorizLineExit

        ; Verifica validade das coordenadas "left" e "right"...
        mov     ax,[left]
        cmp     ax,[right]
        jb      @@noSwap

        ; Troca "left" por "right"
        ;  se "right" for menor que "left".
        xchg    ax,[left]
        mov     [right],ax

@@noSwap:
        ; Verifica a validade das coordenadas "left" e "right"
        cmp     ax,MAX_X_POS    ; "left" é valido?
        ja      @@grHorizLineExit

        or      [right],0       ; "right" é valido?
        js      @@grHorizLineExit

        WriteMode   3     ; Ajusta no modo de escrita 3.
        BitMask     0FFh  ; BitMask totalmente setado!
        MapMask     1111b ; Habilita todos os quatro planos
                          ;  de bits.
        SetReset    <[byte color]> ; Ajusta a cor desejada...

        mov     ax,SCREEN_SEGMENT
        mov     es,ax   ; ES = segmento de vídeo.

        ; Calcula os offsets das colunas...
        mov     si,[left]
        mov     di,[right]
        shr     si,3        ; si = offset da coluna 'left'
        shr     di,3        ; di = offset da coluna 'right'

        ; Calcula o offset da linha 'y'
        mov     bx,[y]
        mov     ax,LINE_SIZE
        mul     bx
        mov     bx,ax   ; BX contém o offset da linha.

        ; Pré-calcula a mascara da coluna 'left'
        mov     cx,[left]
```

```
        mov     ch,cl
        and     ch,111b
        mov     cl,8
        sub     cl,ch
        mov     ah,0FFh
        shl     ah,cl
        not     ah
        mov     [bitmask1],ah

        ; pré-calcula a mascara da coluna 'right'
        mov     cx,[right]
        and     cl,111b
        inc     cl
        mov     ah,0FFh
        shr     ah,cl
        not     ah
        mov     [bitmask2],ah

        ; Verifica se apenas um byte será atualizado.
        cmp     si,di
        jz      @@OneByte

        mov     ah,[bitmask1]
        xchg    [es:bx+si],ah  ; Escreve na memória da video...
                               ; ... XCHG primeiro lê o que
                               ;  está no operando destino,
                               ;  depois efetua a troca.
                               ;  Com isso economizamos um MOV!
        inc     si
        cmp     si,di
        je      @@doMask2

@@MiddleDraw:
        mov     [byte es:bx+si],0ffh    ; Linha cheia...
                                        ; Não precisamos
                                        ;  carregar os latches
                                        ;  pq todos os bits
                                        ;  serão atualizados!
        inc     si
        cmp     si,di
        jne     @@MiddleDraw

@@doMask2:
        mov     ah,[bitmask2]
        xchg    [es:bx+si],ah   ; Escreve na memória de vídeo
        jmp     @@HorizLineEnd

@@OneByte:
        and     ah,[bitmask1]
        xchg    [es:bx+si],ah

@@HorizLineEnd:
        WriteMode 0         ; Poe no modo 0 de novo...
                            ;  Necessário somente se essa
                            ;  rotina for usada em conjunto
                            ;  com as rotinas da BIOS ou de
                            ;  seu compilados (p.ex: BGIs!).
@@grHorizLineExit:
        ret
endp

;;*** DESENHA LINHA VERTICAL ***
proc    grVertLine
arg     x:word, top:word, bottom:word, color:byte
```

```
uses    si, di

        ; Verifica se X está na faixa
        mov     ax,[x]
        or      ax,ax               ; x < 0?
        js      @@grVertLineExit

        cmp     ax,MAX_X_POS        ; x > 639?
        ja      @@grVertLineExit

        ; Verifica se precisa fazer swap
        mov     ax,[top]
        cmp     ax,[bottom]
        jb      @@noSwap

        xchg    ax,[bottom]
        mov     [top],ax

@@noSwap:
        ; Verifica se as coordenadas "Y" estão dentro da faixa.
        cmp     ax,MAX_Y_POS
        ja      @@grVertLineExit

        cmp     [bottom],0
        js      @@grVertLineExit

        mov     ax,SCREEN_SEGMENT
        mov     es,ax

        WriteMode 3
        BitMask 0FFh
        MapMask 0Fh
        SetReset <[byte color]>

        mov     si,[top]

        mov     ax,LINE_SIZE
        mul     si
        mov     bx,ax       ; BX contém o offset da linha

        mov     di,[x]
        mov     cx,di
        shr     di,3        ; DI contém o offset da coluna

        and     cl,111b
        mov     ah,10000000b
        shr     ah,cl

@@SetPixelLoop:
        mov     cl,ah
        xchg    [es:bx+di],cl
        add     bx,LINE_SIZE
        inc     si
        cmp     si,[bottom]
        jbe     @@SetPixelLoop

        WriteMode 0

@@grVertLineExit:
        ret
endp

proc    setGraphMode
    mov     ax,12h
```

```
    int     10h
    ret
endp

proc    setTextMode
    mov     ax,3
    int     10h
    ret
endp

end
```

Não sei se percebeu a engenhosidade dessa pequena rotina... Ela pré-calcula os bitmasks do inicio e do fim da linha... Se a linha está contida somente em um byte então fazemos um AND com os dois bitmasks pré-calculados pra obter o bitmask necessário para atualizar um único byte... Suponha que queiramos traçar uma linha de (2,0) até (6,0). Eis os bitmasks:

```
BitMask1    =   00111111b   ; BitMask do inicio da linha
BitMask2    =   11111110b   ; BitMask do fim da linha
 ---------------------------
BitMask3    =   00111110b   ; BitMask1 AND BitMask2
```

Ok... E se a linha ocupar 2 bytes?! Por exemplo, de (2,0) até (11,0)... O ponto (2,0) está, com certeza, no primeiro byte... mas o ponto (11,0) não (já que um byte suporta apenas 8 pixeis!). Então calculados os dois bitmasks:

```
BitMask1    =   00111111b   ; BitMask do inicio da linha
BitMask2    =   11110000b   ; BitMask do fim da linha
```

Dai escrevemos o primeiro byte com o bitmask1 e o segundo com o bitmask2. Se a linha ocupar mais de 2 bytes o processo é o mesmo, só que os bytes intermediários terão bitmasks totalmente setados (não necessitando, neste caso, carregar os latches!).

Na mesma listagem temos a rotina de traçagem de linhas verticais... dê uma olhada nela. É bem mais simples que grHorizLine!

No próximo texto: O modo de escrita 2! E depois, os modos de 256 cores! (finalmente, né?!)

Vistos os três primeiros modos de escrita da placa VGA, nos resta apenas o modo 2. Esse modo é muito útil para escrita de bitmaps nos modos de vídeo de 16 cores... Ele trabalha basicamente como o registro Set/Reset, sem que tenhamos que manusear esse registro explicitamente.

¦ O modo de escrita 2

Uma vez setado, o modo de escrita 2 habilita todos os quatro bits de "Enable Set/Reset", da mesma forma que o modo de escrita 3. No entanto, diferente do modo de escrita 3, o registro Set/Reset não precisa ser ajustado com a "cor" desejada. Neste modo o registro Set/Reset é setado com os quatro bits menos significativos enviados pela CPU à memória do sistema. Precisaremos mascarar os bits não desejados em BitMask, bem como ajustar os planos de bits desejados em MapMask.

Repare na força deste modo de vídeo... poderemos atualizar pixels com a "cor" que quisermos sem usarmos Set/Reset diretamente, e sem termos que setar os bits de "Enable Set/Reset". Mas, teremos que ajustar BitMask para não setarmos todos os oito pixels no byte que estamos escrevendo dos planos de bits... Eis um exemplo do modo de escrita 2:

```
ideal
model tiny
locals
jumps

include "vga.inc"

LINE_LENGTH     equ     80

codeseg
org     100h
start:
    mov     ax,12h  ; Ajusta modo de vídeo 640x480x16
    int     10h

    WriteMode   2       ; modo de escrita 2
    MapMask     1111b   ; todos os planos de bits

    mov     ax,0A000h
    mov     es,ax       ; ES = segmento de vídeo

    sub     di,di       ; DI = offset
    sub     bl,bl       ; usaremos BL p/ contar as linhas.

    mov     ah,10000000b ; ah = bitmask inicial
    mov     cl,1000b     ; CL = cor inicial

@@1:
    BitMask ah
    mov     al,[es:di]  ; carrega latches
    mov     [es:di],cl  ; escreve nos planos
    ror     ah,1        ; rotaciona bitmask
    inc     cl          ; próxima cor
    cmp     cl,10000b   ; ops... ultrapassou?!
    jb      @@1         ; não... então permanece no loop.
```

```
      mov     cl,1000b    ; ajusta p/ cor inicial.
      add     di,LINE_LENGTH ; próxima linha
      inc     bl          ; incrementa contador de linhas
      cmp     bl,8        ; chegou na linha 8?
      jb      @@1         ; não... continua no loop.

      sub     ah,ah       ; espera tecla, senão não tem graça!
      int     16h

      mov     ax,3        ; volta ao modo texto...
      int     10h

      int     20h         ; fim do programa.
  end start
```

Esse modo parece mais fácil que os demais, não?! Aparentemente é... mas tenha em mente que os outros modos de escrita também têm suas vantagens.

¦ E os modos de leitura?!

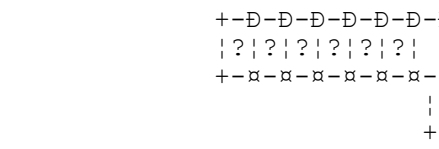
Na grande maioria das vezes não é vantajoso lermos os dados que estão nos planos de bits... Isso porque a memória de vídeo é mais lenta que a memória do sistema (mesmo a memória do sistema associada à placa VGA é mais lenta que o resto da memória do seu PC... por causa dos WAIT STATES que a placa VGA adiciona para não se perder - a velocidade da CPU é maior que a do circuito de vídeo!).

Para encerrarmos os modos de 16 cores é interessante vermos alguma coisa sobre o modo de leitura 0, que é o modo default da placa VGA.

No modo de leitura 0 devemos ler um plano de bits por vez... não é possível ler mais que um plano ao mesmo tempo... e ainda, MapMask não é responsável pela habilitação dos planos de bits. Nesse caso a leitura é feita através de uma ramificação do circuito de vídeo... a escrita é feita por outra. O registrador BitMask também não tem nenhum efeito na leitura. Por isso a seleção dos bits fica por sua conta (através de instruções AND).

A seleção do plano de bits que será lido é feito pelo registrador ReadMap que é descrito abaixo:

» Registrador READMAP



ReadMap também faz parte do circuito GC... Então é acessível via endereços de I/O 3CEh e 3CFh, da mesma forma que BitMask e o registro de MODE, só que seu índice é 4.

Uma nota importante é a de que, embora a leitura seja feita por uma ramificação diferente (por isso a existência de ReadMap), quando fazemos uma leitura dos planos de bits, os latches são automaticamente carregados... e os latches pertencem à ramificação do circuito de escrita (somente os latches dos planos selecionados por MapMask são carregados, lembra?!).

E zé fini... pelo menos até o próximo texto! :)