```
i-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula Nº 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 liguagens 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.

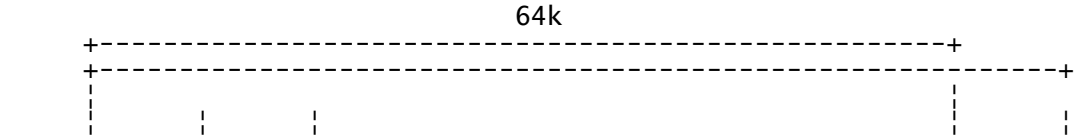
Uma grande desvantagem do assembly com relaçäo as outras linguagens é que näo existe tipagem de dados como, por exemplo, ponto-flutuante... O programador terá que desenvolver as suas próprias rotinas ou lançar mao do co-processador matemático (o TURBO ASSEMBLER, da Borland, fornece uma maneira de emular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

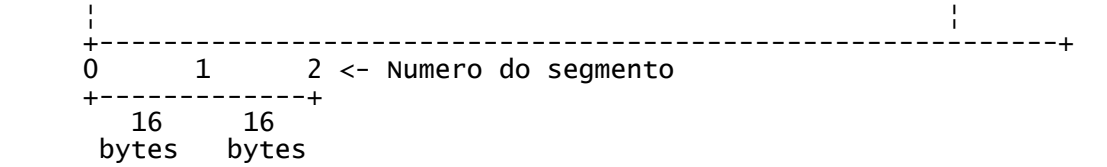
Pra que serve o assembly entäo? A resposta é: Para que você possa desenvolver as suas próprias rotinas, sem ter que topar com bugs ou limitaçöes de rotinas já existentes na ROM-BIOS ou no seu compilador "C", "PASCAL" ou qualquer outro... Cabe aqui uma consideraçäo interessante: É muito mais produtivo usarmos uma liguagem de alto nível juntamente com nossas rotinas em assembly... Evita-se a "reinvençäo da roda" e näo temos que desenvolver TODAS as rotinas necessárias para os nossos programas. Em particular, o assembly é muito útil quando queremos criar rotinas que näo existem na liguagem 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 segmentaçäo ¦
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 näo säo organizados de forma sequencial (o segmento seguinte näo começa logo após o anterior!). Existe uma sobreposiçao. 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, ninguem 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 notaçäo 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 desesseis primeiros bytes do segmento 0 e os últimos desesseis 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 mäos... fica facil contar nos dedos quando precisamos (hehe).

Computadores usam o sistema binário por um outro motimo 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 número 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 representariamos o mesmo númer se tivessemos 16 dedos nas mäos?

¦ Primeiro teriamos que obter mais digitos... 0 até 9 não são suficientes. Pegaremos mais 6 letras do alfabeto para suprir esta deficiencia.

¦ 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 número 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 númer 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 número em hexadecimal. Neste caso, 1994=7CAh

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

O processo inverso, hexa->decimal, é mais simples... basta escrever o númer, 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 número 12 em binário fica:

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

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

A vantagem de usarmos um número 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.