

## O COMEÇO:

Diretamente de um lugar onde nenhum homem jamais esteve, aqui finalmente começa a nossa série imperdível de tutoriais para aqueles, que assim como eu estão aprendendo a programar em Assembly. Mas antes de mais nada para quem diz que Assembly é antigo, fora de moda, não é portátil e difícil... FUCK YOURSELF WITH A BIG AND A HUGE CONSOLE, por que não sabe o que estão dizendo.

Bem como você já devem ter notado não vou poupar palavras para escrever esses tutoriais, e se você tem coração fraco, feche seu processador de textos TXT ou seu navegador de Internet, e volte a sua vida de "Coder to Cash" e suas limitações e claro lentidão imposta pelo seu compilador. Entretanto não reclame quando você for instalar um programa qualquer e este exigir uma maquina dez vezes melhor que a sua "Top de Linha" que você acabou de comprar.

Estarei explicando cada função e instrução da maneira mais detalhada possível, entretanto Bases Numéricas e Tipos de Dados e coisas corriqueiras, eu estarei apenas fazendo uma passagem superficial, por que existem milhões de tutoriais que abordam o mesmo assunto.

A parte mais interessante deste Tutorial, é sem dúvida a parte em que eu explico a Linguagem Assembly, a explicação foi feita junto com a elaboração de programas, sendo eles bem simples apenas para demonstrar como funciona cada função e instrução.

Espero que você leitor goste deste 'Tutorial Base', e se aprofunde mais nesse mundo da verdadeira programação! ☺

## OBSERVAÇÕES:

Algumas coisas já serão colocadas na mesa a partir de agora, para que ninguém se perca quando eu falar: x86, MPC, MIPS, ASM, C e etc. Ai vai a pequena lista... number one:

.ASM: Abreviatura de Assembly

.Assembly \*: A Linguagem de Baixo-Nível. Que compreende em escrever um programa de maneira que um ser humano entenda. Todo código em Assembly deve ser escrito em um editor que NÃO FORMATE TEXTO, (Como o próprio Bloco de Notas do Windows) ou um editor \*.TXT padrão, ou a própria interface do MONTADOR / ASSEMBLER.

.Assembler: É o 'Montador' que transforma o código 'HUMANO' ou seja aquele que nós escrevemos, em algo que o computador entenda ou em linguagem de máquina.

.Coder: Programador.

.Debug: Depurar, e com ele você também pode criar programas .COM em 'DOS'.

.Depurar: Significa verificar o código digitado em busca de erros.

.DOS: Sistema Operacional com interface de linha de comando. Usuários Windows conhecem DOS como MS-DOS.

.MIPS: Millions Instructions Per Second. (Preciso traduzir?)

.MPC: Microprocessador, é o cérebro do computador!

.x86: Arquitetura de processadores IBM/PC compatíveis, 16 e 32 bits.

## O PROCESSADOR :

Uma das coisas mais importantes que você precisa saber em primeiro lugar, é que o processador não compreende nosso sistema de contagem decimal, aliás ele não compreende nada além de uma extensa carreira de 0 e 1.

### COMO ASSIM 0 e 1?

Será que estou ficando louco? - Não para todo tipo de pergunta há uma explicação. Entretanto aqui não é tão fácil de explicar sem a ajuda da sua IMAGINAÇÃO! Mas antes um pouco sobre Números e Bases.

## NÚMEROS BINÁRIOS:

Aprender o que são números binários (O que é relativamente fácil), não é apenas saber que eles são base 2. Pelo contrário ajuda ao iniciante compreender futuros tópicos como: *Complemento de Dois*, *Representação de dados*, *Flags*, *Operações Lógicas*, e assim por diante.

O.k. vamos dançar conforme a música, base 2, base 10 e 16 (Isso não é beisebol). Essas três bases serão aquelas que irá estar sempre ao nosso lado na jornada rumo ao programa perfeito em Assembly.

O que essas 3 Bases tem em comum?

Quase Nada. ☺ Isso mesmo quase nada, talvez em relação a base 2 e base 16 essas bases até poderiam ser primas de terceiro grau, mas em se tratando da base 10 que é a nossa base para uso diário na escola, supermercado, Banco e etc. Essa Base não chega ser nem vizinha das outras duas!

Mas chega de enrolar, vamos entender cada uma dessas bases.

## BASE 10 - DECIMAL:

**BASE 10:** Vamos começar pela base usada em 99,9% do Globo terrestre, aquela que nos orienta quando queremos saber quantos Quilômetros ainda faltam para chegarmos a um determinado lugar, para sabermos o peso de certas coisas e etc.

**Base 10 ou Decimal:** Diferentemente do que algumas pessoas possam pensar, a base 10 não começa em 1 e termina em 10. (O mais incrível é que já deparei com figuras que pensavam assim vagando na Internet).

O Correto é: 0,1,2,3,4,5,6,7,8,9 ou seja temos 10 números ao todo.

*Putz por que sempre se esquecem do 0 zero!?! :(*

O que o torna esse método o mais usado em nosso dia a dia é fato de nós possuímos "10" dedos em nossas mãos, sendo assim torna muito fácil a contagem! SERÁ?

## BASE 2 - BINÁRIO:

**Base 2 ou BINÁRIO:** (significa 2), começa em 0 zero e termina em 1 um!

E adivinha só, essas duas unidades "0 e 1" são as únicas coisas que nosso MPC entende!

Provavelmente isso pode estar começando a ficar um pouco estranho, principalmente se este for seu primeiro contato com Assembly ou mesmo entendendo o funcionamento do MPC, mas há uma luz no fim do Túnel, que você há de encontrar nos próximos tópicos, então se segure um pouco mais na poltrona.

Só uma coisa antes de continuarmos, talvez você esteja com uma dúvida sobre a relação Binária e os Computadores, aliás eu mesmo já tive essa dúvida. Mas então aprendi que a falta de \$\$\$ dinheiro cria saídas realmente interessantes. O fato é que em 1900 e bolinhas, os mestres ou engenheiros da computação

precisavam processar e armazenar dados de maneira mais eficaz possível para o processador, e através de muito estudo e o uso do esquema de Boolean, eles conseguiram matar '10' (hehe '10' valor binário para o 2 decimal) coelhos com uma cajadada só!

Resumindo o computador é uma maquina de estados, e com 2 estado podemos definir o que realmente importa para o computador:

1	0
LIGADO	DESLIGADO
VERDADEIRO	FALSO
HOMEM	MULHER
SIM	NÃO

Esses 2 estados são muito valiosos, sendo que 1 e 0 representam a voltagem 5v e 0v respectivamente. Mas não se prenda muito a isso, e logo entrarei em mais detalhes.

**BASE 16 - HEXADECIMAL:** HEXADECIMAL (significa 16) ! Humm...

Em Base 10 temos: 0,1,2,3,5,6,7,8,9 - Certo? Em Hexadecimal temos esses mesmos números mais '5' caracteres, que compreendem as letras do ALFABETO de 'A' a 'F'.

Então na Base 16 temos: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Ficou um pouco confuso vendo LETRAS se misturando com NÚMEROS?

Veja a tabela de comparação entre DEC, HEX e BIN:

DECIMAL	HEXADECIMAL	BINÁRIO
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

### ATENÇÃO:

As coisas parecem estar meio complicadas mas acalma-se por que elas vão piorar! ☺ Brincadeira (Eu sempre quis falar isso)!

O.k. Note como de uma forma e de outra parece que o Hexadecimal se encaixe perfeitamente no Binário. Se você não entendeu o que eu quis dizer? Então olhe o último número da nossa tabela:

DECIMAL	HEXADECIMAL	BINÁRIO
15	F	1111

O último elemento da Base 16 é igual ao último elemento Binário de 4 Bits!

*BITS: É cada uma dessas unidades de 0 e 1 em Binário.*

Se você olhar a Tabela do início ao fim da 3ª Coluna, pode reparar que ocorre o famoso "VAI UM". EX:

0000 - Nosso primeiro número Binário! Igual '0' DECIMAL.

0001 - Aqui ocorre o seguinte  $0000 + 0001 = 0001$  OK?

0010 - E agora vemos o famoso "VAI UM"  $0001 + 0001 = 0010$ ! Outra ótica:

0001		0001
+0001		+0001
<hr/>		<hr/>
0002 DECIMAL		0010 BINÁRIO

Como só temos 2 Bases em Binário, então nós só podemos representar os números com 0 e 1, esse esquema de "VAI UM" será constante.

Em Decimal o "VAI UM" só acontece após o 10º número da nossa Tabela.

$$\begin{array}{r} 9 \\ +1 \\ \hline 10 \end{array}$$

Mas talvez você possa estar se perguntando. O quê acontece ao último elemento binário da nossa tabela com relação a esse "Vai Um"?

Primeiro vejamos parte da tabela:

DECIMAL	HEXADECIMAL	BINÁRIO
15	F	1111

E agora vamos adicionar '1' ao nosso '1111' Binário.

$$\begin{array}{r} 1111 \\ + 1 - \text{MUITA CONCENTRAÇÃO AGORA!} \\ \hline 10000 \end{array}$$

UAU! Viu só. Se você pegar um lápis e caderno e fazer o "VAI UM", talvez ficará ainda mais fácil de entender o que acontece. E só mais um detalhe, o modo correto de escrever esse número que obtemos na soma seria:

'0001 0000' = Um número de 8 Bits padronizado! (No número acima apenas adicionamos 3 zeros a esquerda).

O.k. Talvez eu tenha mostrado algumas coisas que não deveria ter visto agora, mas lembre-se de continuar a leitura, e qualquer coisa é só voltar ao ponto da dúvida que as coisas ficarão mais claras.

### **PONTO DE DISCÓRDIA:**

Enfim para embaralhar um pouco mais sua cabeça, chegamos ao ponto da “discórdia”!

Ops na verdade eu não queria ter digitado a frase ai em cima, mas não agüentei, por acaso você ficou com medo? ☺ - NÃO DEVERIA.

O quê torna HEXADECIMAL o ponto da discórdia é muito simples, e vale até uma pergunta:

*O que vale mais a pena, ser o Mestre HEXADECIMAL, ou Mestre ASSEMBLY?*

Claro que você deve estar imaginando, por que não ser o MESTRE nas duas coisas, mas por incrível que pareça, nunca através das noites em claro em frente ao PC, e vagando pela Internet encontrei alguém que soubesse HEXA de cor e salteado. E a explicação para isso é muito simples, pois muitas vezes a pessoa que está começando os estudos em "ASM" não quer "perder" tempo com HEXADECIMAL. Mas não posso culpa-los pois por experiência própria, eu sei que isso requer bastante tempo e prática, pois fazer uma conversão de um número DECIMAL para HEXADECIMAL não é apenas somar e subtrair, você tem que Elevar e Multiplicar e por fim Somar tudo, algo que requer gosto pela coisa.

Mas então você deve estar se perguntando, se aprender HEXA é ótimo para trabalhar com números Binários, mas ao mesmo tempo requer bastante tempo e prática, como farei meus primeiros programas já que estou iniciando ASSEMBLY?

*Simples tente alguns desses caminhos válidos para usuários Windows®:*

Iniciar/Programas/Acessórios/Calculadora  
Iniciar/Executar e na linha de comando digite 'Calc.exe'.

O.k. Com a calculadora aberta clique em 'Exibir' e selecione 'Científica' e.... \*\*\*TANAM\*\*\*... uma ótima calculadora DECIMAL, HEXADECIMAL, BINÁRIO e até OCTAL (Base 8).

Faça um teste: digite '10' e aperte F5,F6,F7 e F8 sendo elas:

*F5 = 'A' (É 10 EM HEXADECIMAL).*

*F6 = '10' (É 10 EM DECIMAL).*

*F7 = '12' (É 10 EM OCTAL).*

*F8 = '1010' (É 10 EM BINÁRIO).*

Você pode tanto usar as teclas de atalho F5 até F8 para a base que desejar ou mesmo usar o mouse e clicar nas opções HEX, DEC, OCT E BIN.

Só para concluir, se você quiser aprender HEXADECIMAL, então estude muito pois conhecimento não faz mal a ninguém. Mas se você estiver tendo problemas com isso, continue vendo "ASM", e use a calculadora do Windows, ou outras do TIPO HP, CASIO e etc. que aceitem HEXADECIMAL.

Vale lembrar também que em HEXADECIMAL existe a famosa 'decoreba'. E você não precisará de uma calculadora ou fazer conta para entrar no modo 13 HEXA ou 19 decimal, quando for fazer seu game no melhor estilo 'old-school'! :D

## PROCESSADOR, O RETORNO:

Você pode ter ficado com dúvida no tópico anterior, sobre como um Processador interpreta um programa ou código escrito por um "CODER". Então finalmente chegamos no final do túnel, e veja a LUZ de uma vez por todas.

PERGUNTA: Como um processador consegue exibir um caracter 'A', não monitor?

RESPOSTA: Fácil preste atenção no código abaixo que exibirá o caracter (LETRA) 'A' na tela.

ASM	HEXA	BINARIO
MOV AH, 02	B402	1011010000000010
MOV DL, 41	B241	1011001001000001
INT 21	CD21	1100110100100001
INT20	CD20	1100110100100000

1ª Coluna: "ASM": É o código que em breve você estará ESCRIVENDO.

2ª Coluna: "HEXA": São as instruções da primeira coluna codificado em HEXA.

3ª Coluna: "Binário": É exatamente isso que o seu processador ENTENDE.

Diz a verdade, a primeira vez quando escrevi: que o Processador só compreendia 0 e 1, você achou que fosse loucura não é? Ou pelo menos que isso não fazia muito sentido. :D

Então logo de cara no código acima, você pode notar que cada instrução em Assembly possui um código binário que será interpretado pelo Processador.

*OBS.: Vale a pena notar também que o nosso pequeno código ai em cima, está em 16 bits.*

Ops... Já falei em Bits uma dúzia de vezes e agora em registradores! Então vamos ver cada um deles. (Depois voltaremos a ver Processadores e o código acima!).

## DADOS:

Bits, Nibbles, Bytes, Word e Double Word, são dados que não passam de farinha do mesmo saco. Só que em sacos de tamanhos diferentes. :D

Antes de explicar cada um deles, vou definir umas 'Regras'.

## REGRAS:

NÚMERO 1: Quando você ler um número binário não se esqueça, como já foi mencionado acima, que toda a contagem começa em 0 (zero). Então:

X X X X X X X X = 8 BITS.  
7 6 5 4 3 2 1 0

O primeiro Bit mais a direita está na posição 0, e o Bit mais esquerda está posição 7. O Bit n.º 7 nesse caso ou "O BIT MAIS A ESQUERDA", será de muita importância, logo mais veremos o por que.

NÚMERO 2: Você verá mais a frente que os processadores 80x86 são antigos 'Trabalhadores' 8 Bits, então sendo assim é melhor estender um Número Binário a 4 ou 8 Bits. Ex: Se temos 110 Binário (6 Decimal) a melhor maneira de escreve-lo será:

Acrescentando 'um zero'	Acrescentando 'cinco zeros'
0110	0000 0110
4 BITS	8 BITS

Foram adicionados zeros para deixar o número com cara de 4 e 8 bits!

NÚMERO 3: Separar os Bits é uma ótima forma de organizar um número. E organizar números é algo que fazemos diariamente. Repare só:

1325698573

Agora veja o mesmo número, mas de forma organizada, sendo que a cada três dígitos usaremos a vírgula, para separar as unidades:

1,325,698,573

Muito mais fácil não? A mesma coisa se aplica aos números binários. Só que ao invés de três, serão 4 dígitos e ao invés da vírgula, usaremos um simples espaço em branco.

Veja o Número 65,000 Decimal, em formato binário de 16 Bits:

1111110111101000

Agora vamos organizar o número Binário acima:

1111 1101 1110 1000

O.k. Tudo certo até aqui, então agora sim: Bits.

### **DADOS - BIT:**

A menor unidade binária: é o BIT. Mas esse único Bit não é nenhum *LOBO SOLITÁRIO*. Na verdade com um único Bit podemos representar 2 números distintos: zero '0' e um '1', e com essa capacidade podemos considerá-lo um ser de dupla personalidade. Com seus Altos e Baixos. :)

Com esses Altos e Baixos ele representa, a Voltagem Alta e Baixa, sendo assim a única coisa que o nosso querido Processador entende desde que foi criado. Usualmente refere-se voltagem Alta a 1 (um) e Baixa a 0 (zero), coincidência ou não esses mesmos números são usados na representação Binária!

**BITS UNIDOS JAMAIS SERÃO VENCIDOS!**

Você pode fazer muitas coisas com um único Bit, mas em geral ele é amplamente usado para representar dados Boolean, SIM ou NÃO, VERDADEIRO ou FALSO, sendo que a 1 se tona a carga positiva e 0 a negativa. Entretanto quando você possui uma série ou carreira desses "PEQUENOS BRILHANTES", aí a coisa muda de figura.

### **DADOS - NIBBLE:**

Esse é o tipo de dado que será lembrado apenas por que com ele podemos representar números Hexadecimais de 0 a F, e Decimais de 0 a 15, ou seja Binário de 0000 a 1111. Então Nibbles serão usados para representar dados de no máximo 4 Bits.

Como você pode notar, a única razão do Nibble existir é simplesmente por razões numéricas.

**PRECISAMOS DE MAIS UNIÃO AQUI!** Vamos ver o BYTE, mas antes vou esclarecer mais algumas coisas.

## FAIXA DE ALCANCE:

Uma coisa que você precisa aprender antes de continuar a explicação sobre BYTE é a FAIXA DE ALCANCE ou RANGE! O que diferencia um Tipo de Dado do outro é a sua FAIXA DE ALCANCE!

Suponhamos que eu queira espaço na minha memória para armazenar o número 5 000. Que Tipo de Dado devo usar? Seria Bit ou Nibble e quem sabe Byte?

Para você descobrir a resposta, você precisa antes saber o 'Alcance' de cada Tipo de Dado. Para isso use a seguinte EQUAÇÃO:

$$\text{'BASE' } ^ \text{'NÚMERO DE BITS'} = \text{'ALCANCE'}$$

BASE: No nosso caso Binário ou seja '2'.

'^': Sinal de elevado.

NÚMERO DE BITS: Dependerá do Tipo de Dado, Bits, Nibbles, Bytes e etc.

Exemplos:

Bit: = 1 Bit, então:

$2^1 = "2"$ . Com um BIT podemos representar 2 números exatamente o 0 e 1.

Nibble = 4 Bits, então:

$2^4 = "16"$ . Exatamente os 16 números Hexadecimais, exatamente como já foi visto no Tópico 'Tipo de Dado' - Nibble.

Byte = 8 Bits, então: Opa vou parar por aqui e voltar a falar de Bytes e lá você saberá a resposta.

## DADOS - BYTE:

Chegamos a um dos mais importantes Tipos de Dados do computador: o Byte!

O Byte possui um total de 8 Bits, e tanto as E/S (Entrada e Saída) quando a memória principal são endereços de um Byte. Assim o menor item que pode ser acessado através de um programa, terá pelo menos um byte de tamanho. Na verdade existe uma maneira de fazer acesso menores mas não muito recomendável mascarando Bits. Mas isso será visto bem mais adiante.

X X X X X X X X								= 8 Bits.
7	6	5	4	3	2	1	0	

X X X X				X X X X				= 2 Nibbles
7	6	5	4	3	2	1	0	
High Nibble				Low Nibble				

Vejamos a estrutura de um BYTE:

É Byte é composto por 2 Nibbles, então um Byte requer 2 Dígitos Hexadecimais.

Abra sua calculadora ou faça de cabeça a transformação do número Hexadecimal: 'FF' em decimal, você terá 255 o Alcance máximo de um Byte.

Byte = 8 Bits:

$2^8 = '256'$ . Esse é o nosso número da sorte! :D

OBS.: 255 ou 256? Lembre-se que toda a contagem começa em 0 e vai até 255! ;)



## POR QUE 1 BYTE TEM 8 BITS?

O motivo pelo qual 1 Byte tem 8 Bits é muito simples e a resposta está passando sobre seus olhos. Adivinhou? Não? Sim! ☺

O Byte tem 8 Bits por causa dos "CARACTERES", ou seja 8 Bits é uma quantidade suficiente para se representar todas as letras do Alfabeto. E a partir do momento que descobriram isso, foram criados parâmetros sempre seguindo uma ordem, se você quer mostrar um número de 0 a 15 use o Nibble, agora se você for armazenar um número de 0 a 9 poderá usar tanto o Nibble quanto o Byte. Mas se você quer representar os caracteres do Alfabeto só poderá usar o Byte.

Tudo certo até aqui? O Lance é ler nas entre linhas, se você teve alguma dificuldade, comece tudo de novo e logo você pega o jeito da coisa.

**DADOS - WORD:**

O.k.! Sem perdermos muito tempo aqui, vou apresentar o Tipo de Dado conhecido como Word:

Word = 16 Bits:

$2^{16} = 65\,536$ . Ótima referencia para números inteiros não acha?

Veja a estrutura de uma Word:

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	= 16 Bits.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	= 2 Bytes
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
High Byte								Low Byte								

**DADOS - DOUBLE WORD:**

Acelerando um pouco mais, chegamos ao último Tipo de dados para os processadores de 32 bits, o Double Word. E como o próprio nome já diz trata-se de uma Word Dupla, ou seja se com 1 (uma) Word temos 16 bits, com 2 (duas) Words temos 32 bits.

Double Word = 32 Bits:

$2^{32} = '4.294.967.296'$ . Gigante não?

Veja a estrutura de uma Double Word:

[illegible]

## DADOS - FINALMENTE

Você deve ter sentido que nos 2 últimos Tipos de Dados, eu voei na velocidade da luz para apresenta-los a você, estou certo? Se você já viu alguma apostila ou tutorial sobre Assembly antes, verá que os autores consomem linhas e mais linhas sobre o assunto. A diferença aqui é que eu quero dar a base e depois mastiga-las entre os exemplos que estarão por vir, e ao contrário de muitos autores que explicam vários termos diferentes ao mesmo tempo, esquecem que muitas vezes o leitor do outro lado é um mero iniciante no mundo da programação e pode se confundir facilmente. Eu quero deixa-los bem confortáveis por enquanto, pois eu sei como vocês se sentem, pois a primeira vez que vi 'Tipos de Dados' fiquei perdido, mas depois que você se acostuma com eles, vai ficar tudo mais fácil e você já terá 1% do caminho andado. :D

### COMPLEMENTO DE 2:

A exemplo do tópico anterior, veja um exemplo simples sobre como funciona o meu método de ensino. Agora que você já sabe o Alcance de vários 'Tipos de Dados', vamos ver um detalhe por trás deles:

Tipo de Dado	Bits	Alcance
Byte	8	0 a 255
Word	16	0 a 65 535
Double Word	32	0 a 4.249.967.295

Então se com uma Word eu posso representar qualquer número positivo de '0 á 65 535', como eu poderia representar um número negativo como por exemplo: -22 000 ?

Para isso existe o famoso Complemento de 2 que possibilita a um certo 'Tipo de Dado' armazenar a parte negativa de um número Inteiro ou Real, ou mesmo caracteres extensivos como '@ , ç , Ç' e etc.

Mas isso requer sacrifícios por parte dos Dados então olhe a seguinte Tabela:

Tipo de Dado	Bits	Alcance	Complemento de 2
Byte	8	256	-128 a 127
Word	16	65 536	-32 768 a 32 768
Double Word	32	4.249.967.296	-2.147.483.648 a 2.147.483.647

O quê acontece aqui é muito simples, para trabalharmos com a parte negativa de um número como por exemplo -22 000, temos que 'Setar' em Binário o primeiro Bit mais a esquerda em 1!

Imagine comigo. Se um Byte é igual a 8 Bits temos:

X X X X X X X X	= 8 Bits.
7 6 5 4 3 2 1 0	

Agora um Byte com Bit mais a esquerda para 'Setado' ou 'Sinalizado', ficaria assim:

1 X X X X X X X	= 8 Bits.
7 6 5 4 3 2 1 0	

Então temos Sete 'X' ou 7 Bits para armazenar dados, usando a equação que aprendemos para saber os Alcances dos Dados, temos:

$$2^7 = '128'$$

Pegue esse '128' e subtraia ao Alcance normal de um Byte '255', temos:

<b>BYTE</b>	255
<b>2<sup>7</sup></b>	-128
<b>Total:</b>	<b>127</b>

Então o Alcance de um BYTE SINALIZADO é de '-128 a 127'.

Vamos ver como fica uma Word. Lembrando que a estrutura de uma Word Não Sinalizada é a seguinte:

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	= 16 Bits.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

E com esse Dado podemos trabalhar com Valores de '0 a 65 535'.

Já uma Word SINALIZADA possui a seguinte estrutura:

1	X	X	X	X	X	X	X	X	X	X	X	X	X	X		= 16 Bits.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Tirando o 1, temos 15 'X' ou 15 Bits para armazenar dados. E usando nossa equação para esse número de Bits temos:

$$2^{15} = '32\ 768'.$$

Subtraímos ao alcance normal de uma Word:

<b>WORD</b>	65 535
<b>2<sup>15</sup></b>	-32 768
<b>Total:</b>	<b>32 767</b>

Com o resultado acima, podemos dizer que com uma Word Sinalizada podemos representar números na faixa de:

WORD SINALIZADA: '-32 768 a 32 767'.

Com isso perdemos a capacidade de armazenar números acima de 32 767, mas em compensação podemos armazenar números negativos.

E por fim vejamos como fica a Double Word:

Veja a estrutura de uma Double Word:

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	32 BITS
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	

E com esse Dado podemos trabalhar com Valores de '0 a 4.294.967.295'.

Já uma Double Word SINALIZADA possui a seguinte estrutura:

1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	32 BITS	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2		1

Tirando o 1, temos 31 'X' ou 31 Bits para armazenar dados. E usando nossa equação para esse número de Bits temos:

$$2^{31} = '2.147.483.648'.$$

Subtraímos ao alcance 'Normal' ou 'Não Sinalizada' de uma Double Word:

Com o resultado acima, podemos dizer que com uma Double Word Sinalizada podemos representar números na faixa de:

<b>DOUBLE WORD</b>	4.249.967.296
<b>2^31</b>	-2.147.483.648
<b>Total:</b>	<b>2.147.483.647</b>

DOUBLE WORD SINALIZADA: '-2.147.483.648 a 2.147.483.647'.

1ª OBSERVAÇÃO: É bem provável que você encontre 2 modos diferentes de dizer 'SINALIZADO', são eles o 'SETADO' e o 'SIGNED'. Todos os modos estão corretos.

2ª OBSERVAÇÃO: Também é provável que veja 'NÃO SETADO' e 'UNSIGNED' para os 'Tipos de Dados' 'NÃO SINALIZADOS', o que também está correto.

### SISTEMAS DE ARMAZENAMENTO:

O Computador não seria tão útil senão pudesse armazenar informações, mesmo essas informações sendo temporárias, como é o caso da Memória RAM que perde todos os seus dados assim que o Computador é desligado, diferentemente do Disco Rígido que mantém seus dados sempre salvos. A diferença entre esses sistemas de armazenamento é a velocidade. Mas infelizmente se você quiser muita velocidade terá que desembolsar muito \$\$\$, e mesmo assim logo faltará memória, por que os programas a cada dia que passa ficam maiores e maiores. Por isso hoje em dia é muito fácil encontrar computadores com a seguinte configuração:

Processador 2.0 Ghz.  
128 Mb de Memória Ram.  
40 Gb de Disco Rígido.  
32 Mb de Vídeo.  
Modem de 56 kb.  
Etc. e etc. e etc.

Mas antes de eu comentar a sinistra configuração acima :D, vamos descobrir o que são KB, MB, GB e etc.

### TAMANHOS DE MEMÓRIA:

Você viu no naquele nosso modelo de computador que tinha diferentes especificações como 128MB, 40GB, 56KB.

O que seriam esses KB, MB ou GB?

Na medida que o computador foi evoluindo seu sistema de armazenamento foi crescendo, e o que antes eram dezenas de Bytes, passaram a ser milhões e atualmente bilhões de Bytes, ou seja tinha que ser encontrada uma medida de para exemplificar as unidades da maneira mais fácil possível. Assim como acontece no nosso dia a dia, entre distâncias de objetos onde usamos: Centímetros, Metros, Quilômetros e etc.

No caso dos computadores uma coisa precisa ser destacada. É que todos os números que compõem nosso sistema de medida são 'Pares', isso por que o número de posições em um circuito integrado de memória, é sempre uma potência de 2. Sendo assim a memória será organizada em:  $2 \times 2 = 4$ , então 4 posições diferentes para armazenamento. Seguindo esse raciocínio:

2x2	=	4
2x4	=	8
2x8	=	16
2x16	=	32
2x32	=	64
2x64	=	128
2x128	=	256
2x256	=	512
2x512	=	<b>1024</b>

Vamos dar uma pausa em 1024, pois ele é ponto de partida para nossa segunda unidade de medida, sendo a primeira o Byte formado por 8 bits.

1024 bytes também pode ser escrito como 1KB, sendo:

KB é a sigla para KiloByte.  
Kilo = Mil.

*ATENÇÃO: O computador possui suas peculiaridades, nesse caso não poderia ser diferente, e sendo assim será impossível termos o 1.000 inteiro, assim como 1.000.000 (Como veremos mais a frente). No caso do computador existe a aproximação, então muitas vezes você verá 1KB que na verdade não são 1000 bytes, mas 1024 bytes.*

Então se meu computador tivesse 64kb de Memória RAM em bytes ele teria:

$64 \times 1024 = 65\,536$  Bytes.

Então sempre que você tiver uma unidade em KiloByte e quiser descobrir seu tamanho em bytes é só multiplicar por 1024.

Vamos ver agora nosso terceiro sistema de medida.

Essa talvez sendo a unidade mais conhecida, e tenho certeza que você sabe:

*Que a maioria dos computadores possuem 64MB, 128MB, 256MB ou mais.  
Que existem memórias de vídeo com 32MB, 64MB, 128MB.  
Que um CD possui 650MB e etc.*

Mas o que esse MB quer dizer?

MB é a sigla para MegaByte.  
Mega = Milhão.

Para chegarmos a 1.000.000 em decimal podemos fazer  $1000 \times 1000$ . Mas no caso dos computadores sabemos que o Kilo é um pouco diferente, então multiplicamos por:

$1024 \times 1024 = 1.048.576$

Silvio Santos que não ia gostar muito desse sistema em seu Show do Milhão não acham? 1MB = 1 048 576

Então agora é só pegar o tamanho da sua Memória RAM, da sua Memória de Vídeo, do seu CD e etc., e multiplicar por 1024 para saber o tamanho em kiloBytes ou 1.048.576 para saber o tamanho em Bytes.

## KiloByte

128 x 1024 = 131 072 KiloBytes

256 x 1024 = 262 144 KiloBytes

650 x 1024 = 665 600 KiloBytes

## Byte

128 x 1048576 = 134 217 728 Bytes

256 x 1048576 = 268 435 456 Bytes

650 x 1048576 = 681 574 400 Bytes

## Legal não?

O nosso quarto e último do nosso grupo de sistemas de medida por enquanto é o GB.

O GB é muito conhecido por causa dos HD (Hard Disk / Disco Rígido). Mas já faz parte da Memória RAM de milhares de grandes empresas pelo mundo inteiro, e de alguns usuários que curtem ter o PC tipo TOP.

KILO é fichinha? Mega não é tão grande assim? E que tal:

GB é a sigla para GigaByte.

Giga = Bilhão.

Ou seja 1 Bilhão de Bytes a sua disposição, parece muito não? Mas realmente não é, Jogos, Programas e Filmes, Músicas são grandes consumidores de memória e muitos deles já beiram aos GigaBytes da vida.

Para chegarmos a 1 Bilhão usamos a seguinte equação: MILHÃO x MIL = RESULTADO.

Mas para acharmos o numero de Bytes absoluto:

MEGA X KILO = RESULTADO (GIGA).

1048576 x 1024 = 1 073 741 824

ou

2<sup>30</sup> = 1 073 741 824

Então seu Disco Rígido de 20GB possui na verdade:

20 x 1024 = 20 480 MegaBytes

20 x 1 048 576 = 20 971 520 KiloBytes

20 x 1 073 741 824 = 21 474 836 480 Bytes

Uma outra forma ao invés de multiplicar por diferentes bases como: 1024, 1 048 576 e 1 073 741 824, para se achar tamanhos em Bytes, KiloBytes e MegaBytes é multiplicar por 1024 várias vezes:

GB		MB		KB		BYTES
20	x 1024 =	20480	x 1024 =	20 971 520	x 1024 =	21 474 836 480

## **ARMAZENAMENTO DE DADOS:**

Uma coisa que é preciso entender agora é como e onde certos tipos de dados são armazenados!

Armazenamentos em Disco Rígidos, Disquetes, CDs, são armazenamentos dinâmicos, depois que salvos os dados ficam ali armazenados pelo tempo que o usuário assim desejar. Mas o grande problema com o armazenamento dinâmico é a sua lentidão na transferência de dados, pois será gasto processamento para transferência desses dados do Disco Rígido para a Memória RAM e depois da Memória RAM para a Cache, e aí sim o processador finalmente poderá trabalhar com esses dados.

Armazenamentos em Registradores, Memória RAM, Cache, são armazenamentos voláteis, ou temporários, ou seja mesmo que o usuário deixe os dados na Memória RAM, assim que ele desligar o computador, esses dados são perdidos. A grande vantagem do armazenamento volátil é a alta velocidade com o qual são trocados os dados com o processador. Entretanto existe um problema quanto a Memória RAM pois esse tipo de memória é muito cara, e por isso muitos computadores são vendidos com uma quantidade bem limitada dessa memória. Hoje em dia 128MB faz parte de uma configuração básica de um computador, mas quase 60% desses 128MB são ocupados pelo Sistema Operacional assim que o computador é ligado. Por isso saber utilizar a Memória RAM é fundamental para o programador que busca alto desempenho, e Assembly é uma ótima maneira de se buscar ALTO DESEMPENHO.

Em se tratando de velocidade das memórias voláteis podemos seguir a linha de raciocínio da tabela abaixo:

**REGISTRADORES:** É a memória interna do Processador. Fica localizado dentro do Processador, por isso é o tipo de memória mais rápida do computador, porém possui apenas alguns Bytes.

CACHE

**MEMÓRIA CACHE:** É a memória que recebe os dados da Memória RAM e envia para os REGISTRADORES, então é o segundo tipo de memória mais rápida é a CACHE, mas possui apenas alguns KiloBytes.

### **MEMÓRIA RAM**

Mais lenta que os REGISTRADORES e a CACHE, entretanto pode armazenar muito mais quantidades de dados. Pode variar entre MegaBytes e GigaBytes, isso dependerá apenas do usuário.

Tudo o que foi visto até aqui são dados técnicos, estou tentando exemplificar da melhor maneira possível, mostrando a porta para você, e se você quiser entrar ou seja saber mais detalhes, existem ótimas referências na Internet.

Agora vamos ver como funciona o 'Debug' nosso primeiro Montador para DOS.

## **DEBUG:**

Debug é o montador ASM presente em computadores com o sistema DOS. Com o Debug você pode fazer programas com no máximo 64KB (ou seja 65 535 Bytes) de tamanho. Sendo assim você não irá precisar saber nada sobre segmentação para começar a fazer programas com ele. (Uma nova segmentação começa a cada 64KB mas isso veremos tarde).

Vamos ao que interessar, encontrar esse 'DEBUG':

Clique em 'Iniciar' e depois em 'Executar':

Digite: 'Debug.exe' e pressione 'Enter'!

Opa uma janela do tipo Prompt-Dos com um hífen surgiu na sua frente?

Muito Prazer esse é o 'DEBUG'.

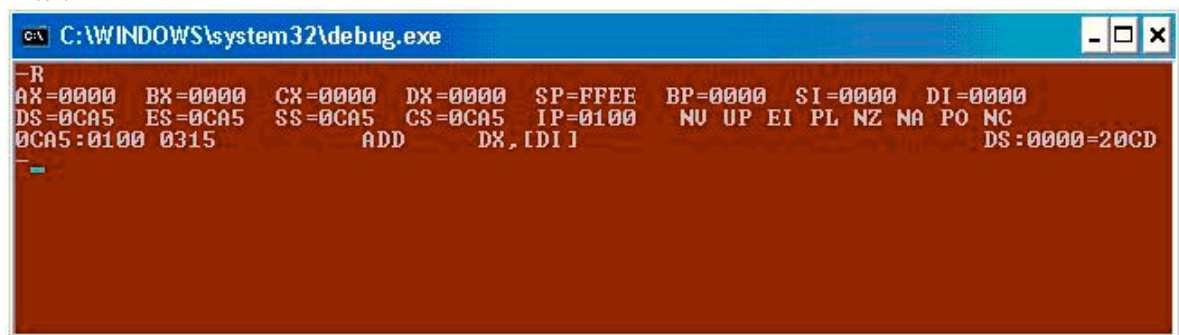
*Obs.: Para sair do 'DEBUG', digite 'Q' e pressione ENTER.*

## REGISTRADORES:

Como você viu no último Tópico, Registradores são memórias internas do Processador, e saber trabalhar com eles é fundamental para uma ótima otimização do seu programa. Um Processador possui um número padrão de Registradores, mais os exclusivos. Nesse Tutorial trabalharei apenas com os Registradores Padrão, para ver os Registradores Exclusivos você deve visitar o site do fabricante do seu processador.

Os Registradores são muito conhecidos como SUPER VARIÁVEIS, elas são como as variáveis da Linguagem C, mas possuem mais poderes.

Vamos ver as variáveis mais de pertinho. Para isso inicie o 'DEBUG', e após o hífen, digite R e pressione Enter:



```
C:\WINDOWS\system32\debug.exe
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CA5 ES=0CA5 SS=0CA5 CS=0CA5 IP=0100  NU UP EI PL NZ NA PO NC
0CA5:0100 0315      ADD     DX,[DI]
DS:0000=20CD
```

Ai estão os Registradores, preste atenção nos 4 primeiros registradores AX,BX,CX,DX, esses quatro registradores estão seguidos por '0000' em Hexadecimal, portanto lembrando a você que a cada dígito Hexa representa 4 Bits, portanto temos em nossa frente quatro registradores de 'propósito geral' com 16 Bits cada.

Outra coisa a ser destacada é que cada um desses 4 Registradores possui a parte ALTA e BAIXA de um BYTE ou seja AX é o conjunto de AH e AL sendo que tanto o AH e AL possuem 8 Bits cada. AX é mesmo que AH + AL. É importante saber isso porque em determinados momentos será necessário trabalhar só com:

'AH' ou com 'AL'.

H=HIGH,ALTO

L=LOW,BAIXO.

O termo propósito geral significa que alguns Registradores podem ser usados para desempenhar 2 ou mais funções diferentes, ao contrário dos Registradores SP,BP,SI,DI,DS,ES,SS,CS,IP que possuem papéis definidos em Assembly.

Estarei descrevendo cada Registrador na medida que forem sendo usados.

## INTERRUPÇÃO:

**INTERRUPÇÃO EM ASSEMBLY:** Ocorre quando a atual atividade é interrompida, passando para uma rotina especial.

**'INTERRUPÇÃO VIA S/O':** É quando o controle é do programa é passado para a INT do Sistema Operacional, a função a ser tomada pela INT é definida pelo programador. O exemplo do uso de uma INT poder ser: 'Imprimir caracteres no Monitor', 'Abrir um Arquivo', 'Sair de Programa' e etc.

**'INTERRUPÇÃO via BIOS':** Está ligado ao Hardware (componente interno do computador), sendo assim é menos portátil por que usa funções específicas da arquitetura em questão, porém é mais rápido que a INTERRUPÇÃO via S/O'.



Porém vamos nos concentrar primeiramente na 'INTERRUPÇÃO VIA S/O', para pegarmos primeiramente a prática da programação.

### **PRIMEIRO PROGRAMA:**

Depois de tanto ler, já está mais do que na hora de botar as coisas em prática não acha? :) Para isso veremos outra vez um código que já apareceu nesse tutorial, quando eu falava sobre o PROCESSADOR. O código era:

```
MOV AH, 02h
MOV DL, 41h
INT 21h
INT 20h
```

No código acima temos:

2 MOV.  
2 Registradores diferentes: AH, DL.  
2 INT Diferentes 21h, 20h.

### **Agora vamos ver como funciona cada um deles:**

**MOV:** É um mnemônico com significado de MOVE/MOVER. Mnemônicos é algo como 'Ajudar a Memória', pois é bem mais fácil uma pessoa se lembrar da palavra MOV do que o código hexadecimal ou binário da mesma função.

Em se tratando de mnemônicos além de MOV em Assembly temos:

```
ADD    Adicionar
SUB     Subtrair
MUL     Multiplicar
DIV     Divisão

INC     INCREMENTAR
DEC     DECREMENTAR

INT     INTERRUPÇÃO
```

Entre outros que veremos depois.

### **MOV = MOVER OU COPIAR?**

Incrível como em milhares de tutoriais e livros você encontra a referência MOVER para MOV.

Mover é tirar Algo de um lugar e colocar em outro. Mas se você fizer `MOV AX, BX` o que acontecerá aqui é uma 'Cópia' de BX para AX. Pois se você ver o conteúdo dos dois Registradores AX e BX após o MOV verá que ambos possuem os mesmos valores. Exemplo:

```
MOV BX, 10
MOV AX, BX
```

No final desse MOV ambos os Registradores possuirão 10.

A Função: `MOV AH, 02h` nada mais faz que copiar 02 em AH. É o mesmo que:

```
MOV destino, fonte
```

Ou seja informação a ser 'COPIADA' no 'DESTINO' vem de onde uma 'FONTE'!

Isso acontece também com o `MOV DL, 41h` que é a copia de 41h para o Registrador DL.

### ATENÇÃO:

Você não pode fazer um `MOV AH, BX` por que nesse caso você estaria copiando BX (16 Bits) em AH (8 Bits) iria exceder a memória.

Tudo bem até aqui? Vamos continuar analisando o código:

### INT 21:

A `INT 21` quando executada checa os Registradores em busca de informação para saber o que deve ser feito. E ao ler o Registrador `AX, 02h`. A `INT 21h` é informada que irá trabalhar com 'Exibição de Caractere'. O caractere a ser exibido está no registrador DL.

`DL, 41` Aqui é feito a copiado do valor o valor HEXA 41h (65 Decimal), que é um código numérico para 'A' em ASCII.

*OBS.: ASCII É uma tabela de códigos que define todos caracteres do Alfabeto, além números e Caracteres especiais como @, #, \$, %.*

Após a Exibir do Caractere 'A', a `INT 21` termina e o código volta a ser executado, mas logo vem outra `INTERRUPÇÃO`, a `INT 20h`.

A `INT 20h` 'Termina a Execução do Programa', e da o comando ao Sistema Operacional.

### DÚVIDAS:

Talvez você tenha algumas dúvidas, mas a principal é como eu sabia que AH tinha que conter 02h e DL 41h, para exibir o caractere 'A' na tela do Computador?

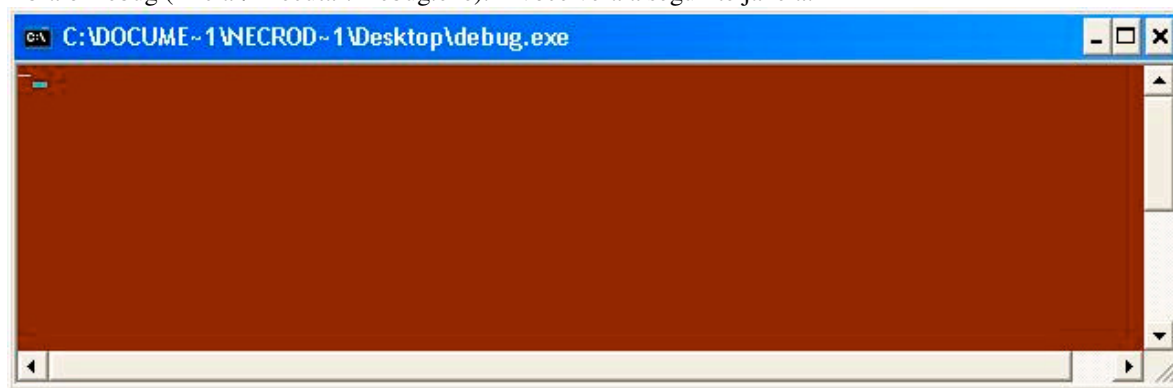
Para isso eu usei um programa chamado HELPPC 2.1 que é o guia para muitas funções em DOS. Ou seja se você quiser saber como 'Ler um Caractere' pelo teclado, ou 'Plotar Pixels' na tela do seu PC, você terá que ter esse guia em seu Computador para fazer consultas.

Você pode fazer o download do Helppc 2.10 em nosso site!

O.K. Espero que você tenha entendido o programa acima, então vamos Monta-lo através do Debug.

### INICIANDO O DEBUG:

Abra o Debug (Iniciar/Executar: Debug.exe). E você verá a seguinte janela:



Estamos no Debug, A primeira coisa em qualquer programa .COM é montar o Endereço, no nosso caso para mantermos o padrão usamos: -A 100 em seguida ENTER.

E Você verá uma resposta parecida com essa.



```
C:\> C:\DOCUME~1\NECROD~1\Desktop\debug.exe
-A 100
0CA5:0100
```

- 0CA5:0100 - O QUE É ISSO?

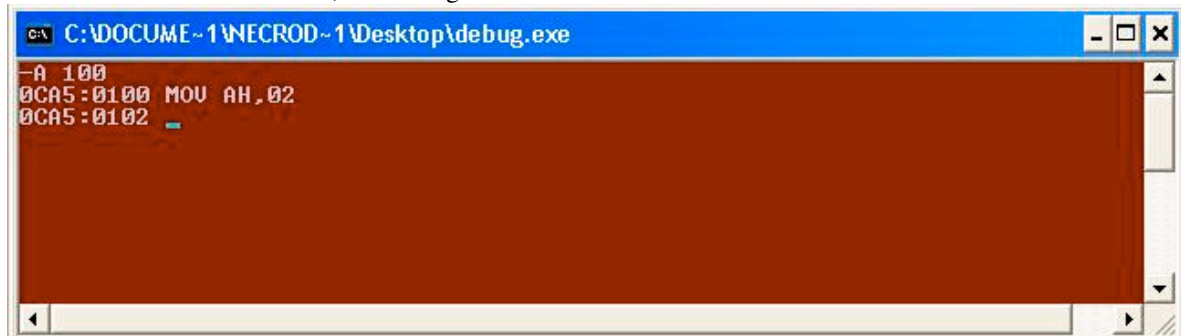
A memória em DOS possui 'segmentos' de 64KB, a cada 64KB começa um 'segmento' novo. Nesse caso o segmento é 0CA5, e o deslocamento dentro segmento encontra-se em 0100.

#### OBSERVAÇÃO:

1º O valor do segmento 'vária' ou seja não é 'fixo'. No exemplo acima o valor do segmento foi '0CA5' mas no seu PC pode ser '0C1B' ou outro número!

2º Não se preocupe com SEGMENTO e DESLOCAMENTO por enquanto, pois programas .COM só possuem um segmento. Mas em breve veremos como cada um deles funciona.

Vamos DIGITAR o -MOV AH, 02 em seguida ENTER.



```
C:\> C:\DOCUME~1\NECROD~1\Desktop\debug.exe
-A 100
0CA5:0100 MOV AH,02
0CA5:0102
```

Repare que esse MOV consumiu 2 Bytes. Olha lá de 0100 passamos para 0102.

Vamos ao segundo MOV aquele que irá informar com qual CHARACTER iremos trabalhar, -MOV DL, 41 seguido do ENTER.

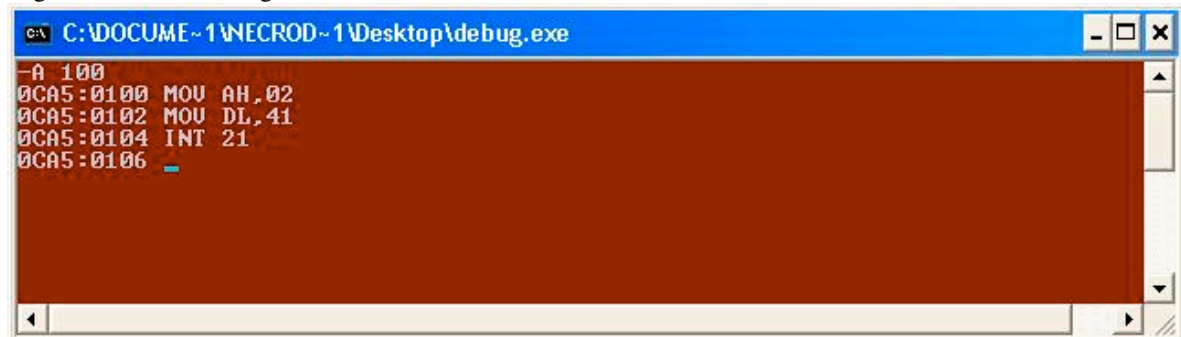


```
C:\> C:\DOCUME~1\NECROD~1\Desktop\debug.exe
-A 100
0CA5:0100 MOV AH,02
0CA5:0102 MOV DL,41
0CA5:0104
```

Mais 2 Bytes foram consumidos. 41H é igual 65 Decimal que é ao caracter 'A' da tabela ASCII.

Chegou a hora de usar a INT que terá o trabalho de percorrer os Registradores em busca de informações, para saber qual Função ela deverá executar.

Digite -INT 21 em seguida ENTER.



```
C:\> C:\DOCUMENTOS\WECROD\1\Desktop\debug.exe
-A 100
0CA5:0100 MOV AH,02
0CA5:0102 MOV DL,41
0CA5:0104 INT 21
0CA5:0106 -
```

Mais 2 Bytes e ai está a estrutura do programa. Eu sei que falta mais uma INT a de saída, mas observe esse código mais um pouco. Quebra-cabeças quase montado não?

Última instrução a -INT 20.

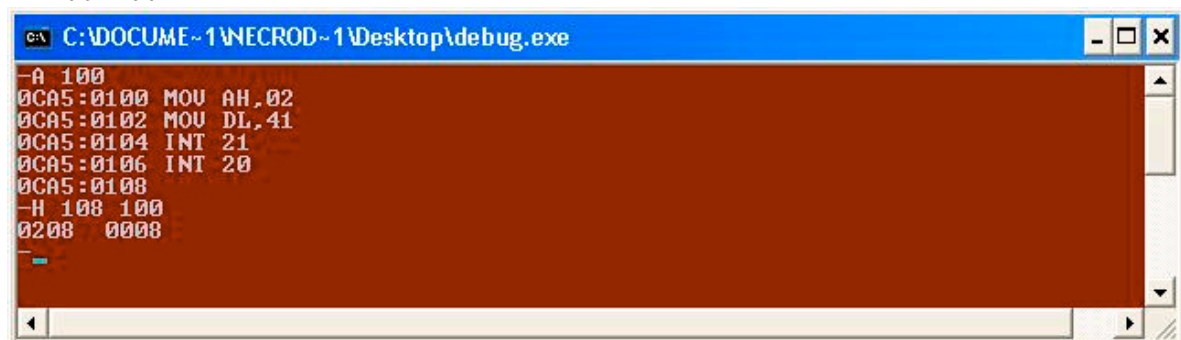


```
C:\> C:\DOCUMENTOS\WECROD\1\Desktop\debug.exe
-A 100
0CA5:0100 MOV AH,02
0CA5:0102 MOV DL,41
0CA5:0104 INT 21
0CA5:0106 INT 20
0CA5:0108 -
```

Agora vamos ao GRAN-FINALE:

Para isso agora sem digitar nada pressione ENTER mais uma vez e o sinal de hífen aparecerá, com isso chegou a hora de pegar o tamanho do nosso programa, subtraindo 0108 de 0100 com o comando:

-H 108 100



```
C:\> C:\DOCUMENTOS\WECROD\1\Desktop\debug.exe
-A 100
0CA5:0100 MOV AH,02
0CA5:0102 MOV DL,41
0CA5:0104 INT 21
0CA5:0106 INT 20
0CA5:0108
-H 108 100
0208 0008
-
```

Você terá duas respostas,

0208 que é a soma de 108 + 100.

0008 que é a subtração de 108 - 100. Você poderia ter feito isso de cabeça não? :D

Agora Vamos nomear nosso programa com o comando -N + nome\_do\_programa.com:

-N TESTE.COM



```
C:\> C:\DOCUMENTOS\WECROD\Desktop\debug.exe
-A 100
0CA5:0100 MOV AH,02
0CA5:0102 MOV DL,41
0CA5:0104 INT 21
0CA5:0106 INT 20
0CA5:0108
-H 108 100
0208 0008
-N TESTE.COM
-
```

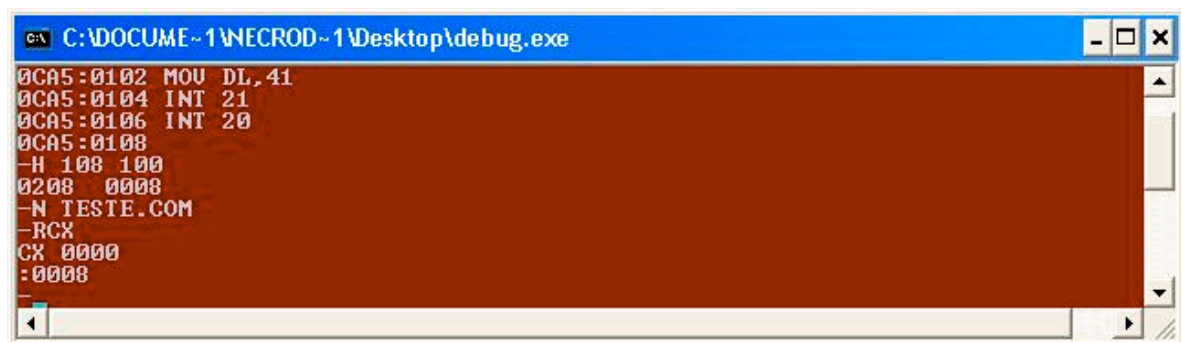
Nomeado o programa digite R + o nome do REGISTRADOR para ter acesso ao mesmo, ter acesso significa poder alterar o seu valor, e nesse caso precisamos fazer isso com o REGISTRADOR CX, onde especificaremos o tamanho do nosso programa com o resultado encontrado na Subtração com H 108 100!

Então digite após o hífen:

-RCX (pressione ENTER).

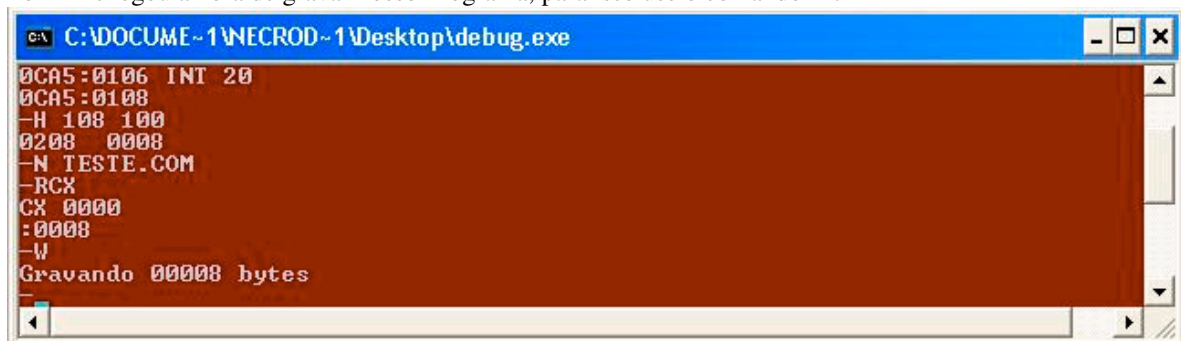
E Depois após os : o tamanho do programa:

:0008 (pressione ENTER).



```
C:\> C:\DOCUMENTOS\WECROD\Desktop\debug.exe
0CA5:0102 MOV DL,41
0CA5:0104 INT 21
0CA5:0106 INT 20
0CA5:0108
-H 108 100
0208 0008
-N TESTE.COM
-RCX
CX 0000
:0008
-
```

Por fim chegou a hora de gravar nosso Programa, para isso use o comando -W:



```
C:\> C:\DOCUMENTOS\WECROD\Desktop\debug.exe
0CA5:0106 INT 20
0CA5:0108
-H 108 100
0208 0008
-N TESTE.COM
-RCX
CX 0000
:0008
-W
Gravando 00008 bytes
-
```

É isso aí terminamos o nosso primeiro programa! No caso de usuários do WindowsXP, seu programa poderá ser encontrado na pasta C:\WINDOWS\SYSTEM32. Ou simplesmente use o comando de Pesquisar/Procurar do Windows.

## INFORMAÇÕES EXTRAS:

Durante a criação do programa usamos o Registradores AH e DL, então deixa eu dar uma revisada nos Registradores:

AX: É conhecido como Registrador ACUMULADOR, esse é o Registrador que você usará para diferentes ocasiões como Funções Matemáticas, ou indicar uma Função e etc.

BX: É conhecido como Registrador de BASE. Por exemplo em uma soma você vai precisar de uma Base, aí é só usar BX

CX: É conhecido como Registrador de CONTAGEM. No caso de um LOOP (Informações que são repetidas por certo período de tempo), o CX será incrementado ou decrementado.

DX: É conhecido como Registrador de DADOS, no programa anterior usamos DL para armazenar um Dado (na ocasião era o caracter 'A'), mas poderia ser uma cadeia de palavras, uma imagem e etc.

*OBS.: O importante é você não misturar os Registradores, como por exemplo usar o Registrador CX para fazer uma SOMA, pois talvez você irá precisar dele para um LOOP (LAÇO), e por fim terá que gastar um MOV para copiar o conteúdo de CX! Para a tal soma acima você pode usar o AX com o BX por exemplo.*

## UM NOVO PROGRAMA:

Agora vamos ao que interessa o LECHAR em Assembly, só que desta vez usaremos um novo montador, o 'FASMW - Flat Assembler for Windows'. A mudança do Debug para o FASM se deve simplesmente por questões de facilidade, mas se você quiser poderá continuar usando o Debug normalmente, já os usuários que querem rapidez na construção de seus programas, o 'FASMW' é uma ótima opção e seu uso é muito simples, já que ele se parece com um bloco de notas, mas com opção de Montar um Programa. Você pode pegar o 'FASMW' aqui mesmo em nosso site na sessão downloads. (São apenas 586KB).

Depois é só descompacta-lo em algum lugar no seu computador. Não precisa de instalação e depois é só clicar em FASMW.EXE e pronto você já estará com o FASMW funcionando.

## DETALHES SOBRE O FASMW:

*O FASMW é um Montador 16 e 32 Bits com sintaxe INTEL.*

*Requer no mínimo um computador com processador Intel 80386 os antigos PCs '386'.*

*A versão para Windows requer Win32 GUI presente a partir do sistema operacional Windows 95.*

*Por padrão ele gera programas em 16 bits, mas o você pode escolher entre 16 e 32 Bits através das Diretivas use16 ou use32.*

Uma propriedade bem legal do FASMW, e muito conhecido por programadores das Linguagens C e C++ são as 'Linhas de Comentários', que ajuda o programador a entender certas funções que ele mesmo criou. Um exemplo bem simples:

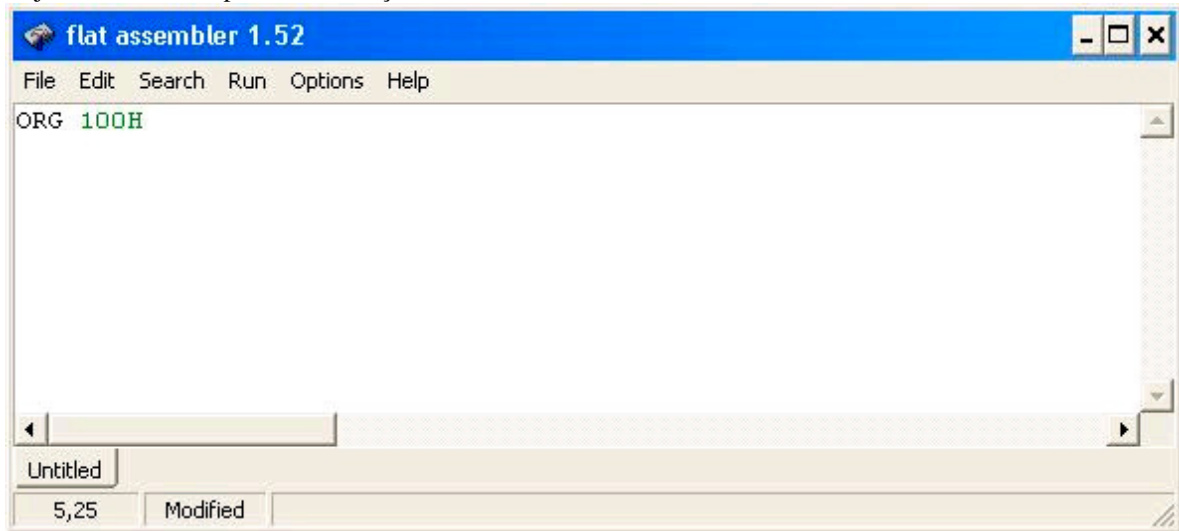
```
MOV AH, 02H ; Copiando 02H em AH para trabalhar com Caracteres.
MOV DL, 41H ; Copiando 41H = 'A' para DL.
INT 21      ; Irá ler AH e imprimir o conteúdo de DL na tela do computador.
INT 20      ; Retorna ao Sistema Operacional.
```

Tudo o que estiver depois de ';' (ponto-e-vírgula) será ignorado pelo compilador.

## O NOVO PROGRAMA - LECHAR:

Vamos dar início ao novo programa e como estaremos fazendo um programa .COM nos moldes do Debug, temos que primeiramente Alocar os 100 bytes iniciais, você deve se lembrar que no Debug nós usávamos o comando 'A 100' para dar início a montagem do programa, no FASMW usamos 'ORG 100' e a partir daí escrevemos o código.

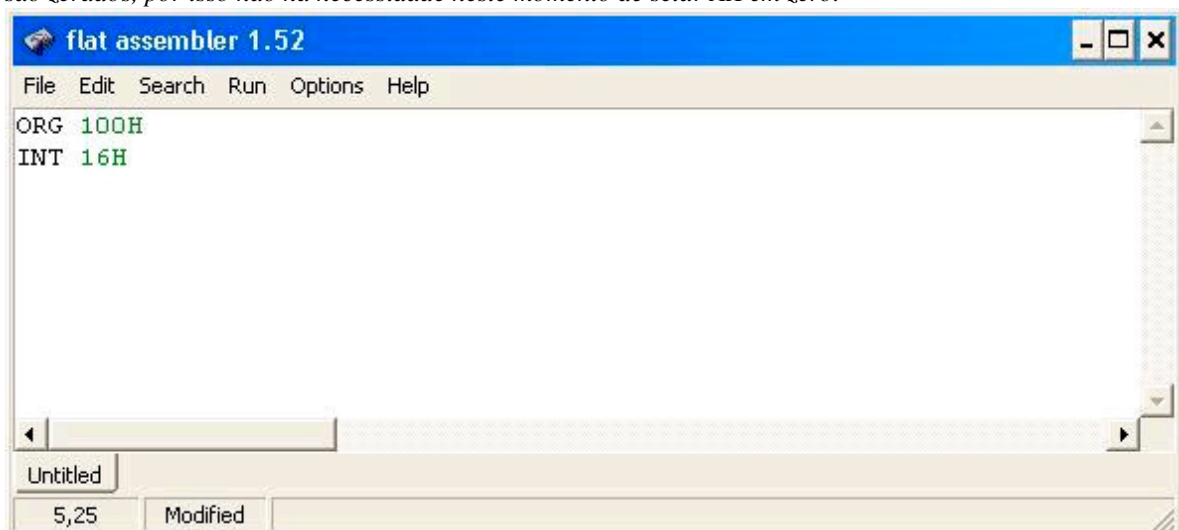
Veja o FASMW e a primeira instrução 'ORG 100':



Note como a primeira vista o FASMW parece o Bloco de Notas.

Chegou a hora de conhecer a Interrupção que irá pegar o caracter digitado pelo usuário, a 'INT 16'. A 'INT 16' com o Registrador AX setado em zero (0000H), interrompe o programa e aguarda até que uma tecla seja pressionada, quando uma tecla for pressionada a 'INT 16' guarda o código do caracter no registrador AL.

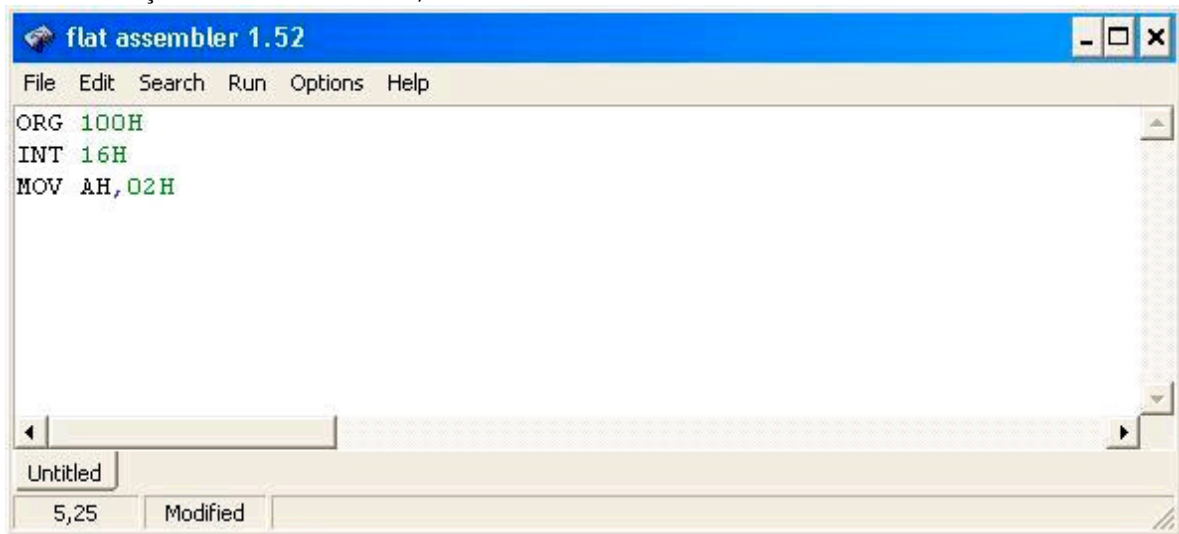
*OBSERVAÇÃO: Quando você executa um programa pela primeira vez, automaticamente os REGISTRADOS são zerados, por isso não há necessidade neste momento de setar AX em zero.*



Note como foi simples essa entrada, pois não foi preciso setar nenhum registrador, por que eles já estão setados com zero.

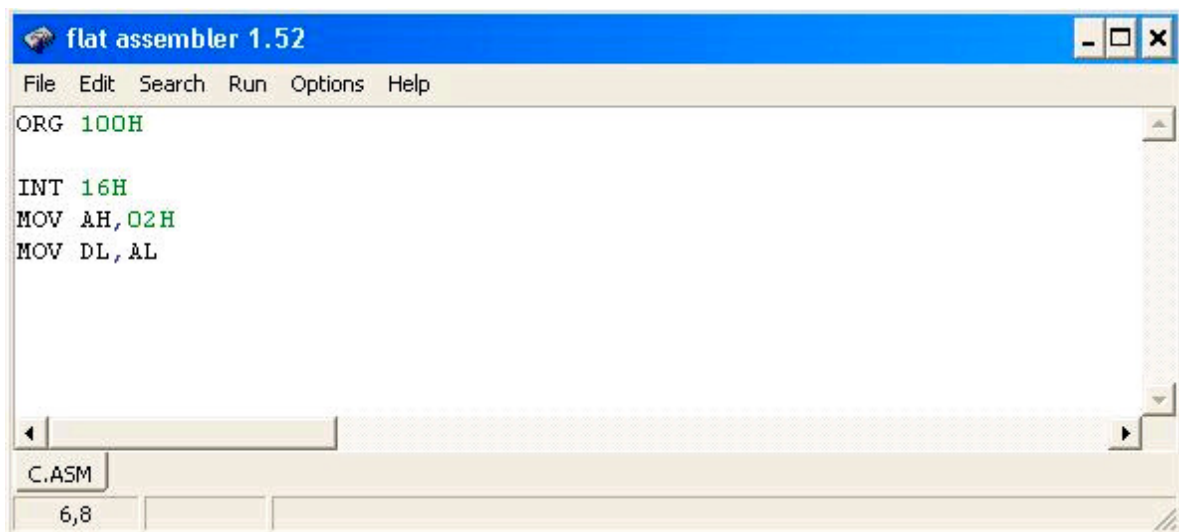
Agora vem a parte mais fácil, pois iremos usar as mesmas instruções que usamos para imprimir o caracter 'A' na tela do computador do programa anterior.

Vamos começar então como 'MOV AH, 02H':



The screenshot shows the 'flat assembler 1.52' window. The menu bar includes 'File', 'Edit', 'Search', 'Run', 'Options', and 'Help'. The main text area contains the following assembly code:  
ORG 100H  
INT 16H  
MOV AH, 02H  
The status bar at the bottom shows 'Untitled', '5,25', and 'Modified'.

A 'INT 21' pega o 'código do caracter' que será mostrado na tela do computador no registrador DL, e como já disse antes a INT 16 salva o 'código do caracter' no registrador AL, então é só copiarmos conteúdo de AL para DL.

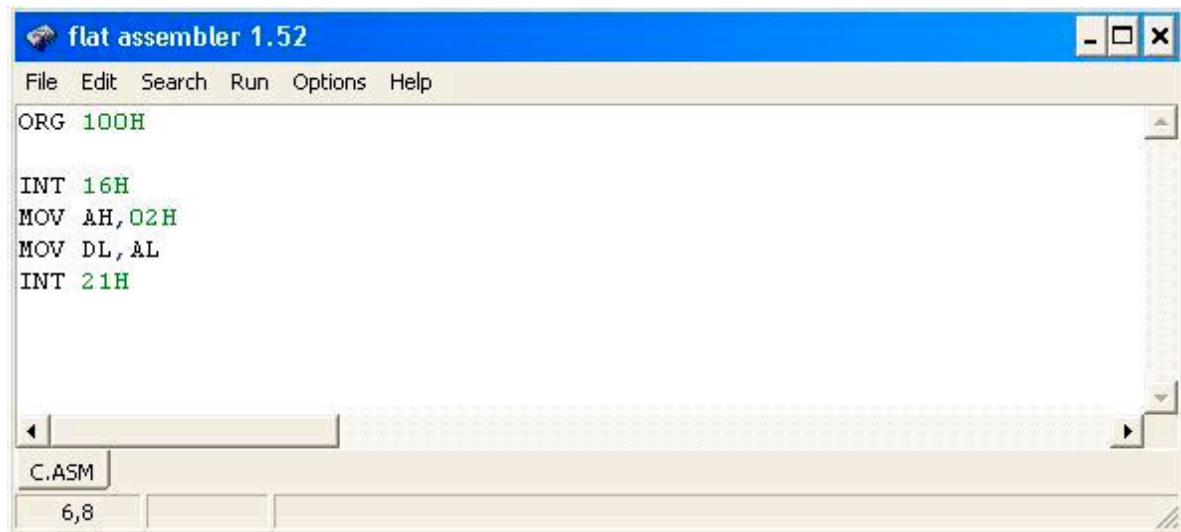


The screenshot shows the 'flat assembler 1.52' window with the same menu bar. The main text area now contains the following assembly code:  
ORG 100H  
  
INT 16H  
MOV AH, 02H  
MOV DL, AL  
The status bar at the bottom shows 'C.ASM', '6,8', and an empty field.

Para isso utilizamos a instrução 'MOV DL, AL'



Vamos colocar a 'INT 21' para nos ajudar a desenvolver nossa função:



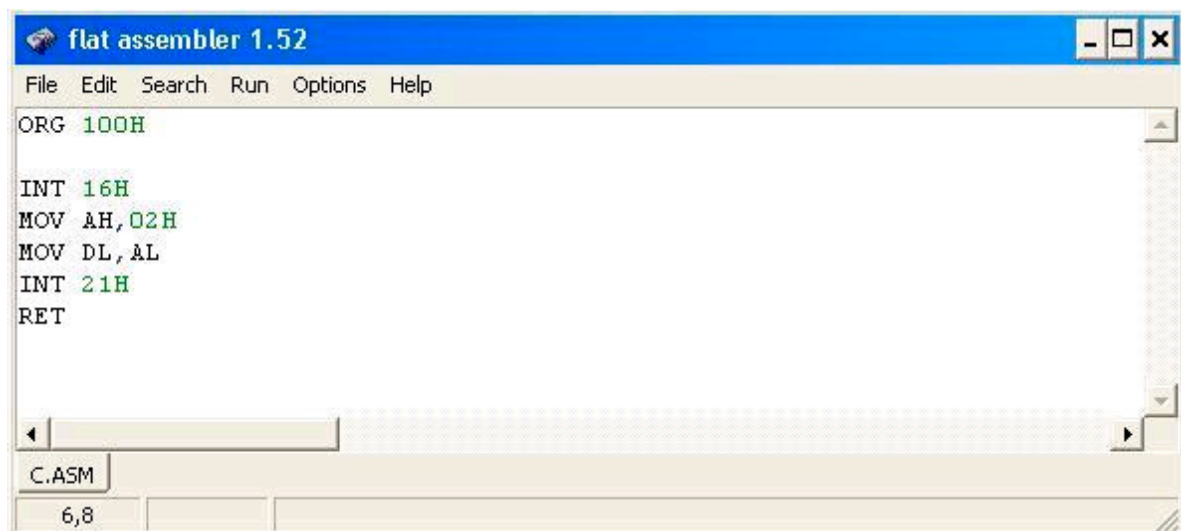
The screenshot shows the 'flat assembler 1.52' window. The menu bar includes 'File', 'Edit', 'Search', 'Run', 'Options', and 'Help'. The main text area contains the following assembly code:

```
ORG 100H

INT 16H
MOV AH, 02H
MOV DL, AL
INT 21H
```

Below the text area is a status bar with a tab labeled 'C.ASM' and a value '6,8'.

Descobri uma coisa interessante no final do programa, e claro a primeira otimização, como você deve saber ainda falta mais uma INT que é a de saída do programa, lembrando que no nosso último programa usamos a INT 20, desta vez será diferente e usaremos o RET (RETORNO) e ganhamos 1 byte com isso. :)



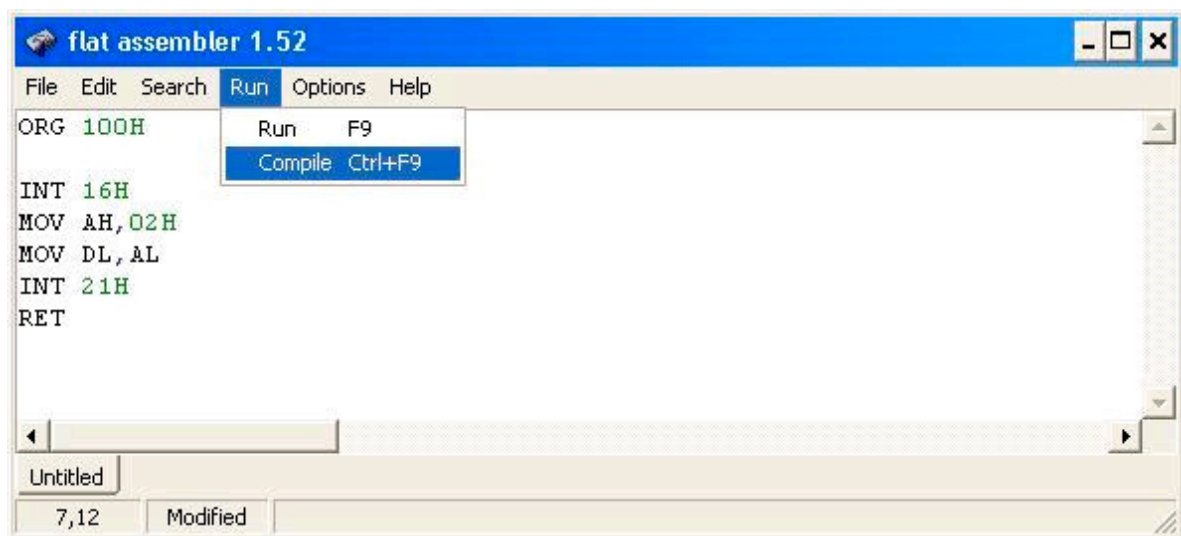
The screenshot shows the 'flat assembler 1.52' window. The menu bar includes 'File', 'Edit', 'Search', 'Run', 'Options', and 'Help'. The main text area contains the following assembly code:

```
ORG 100H

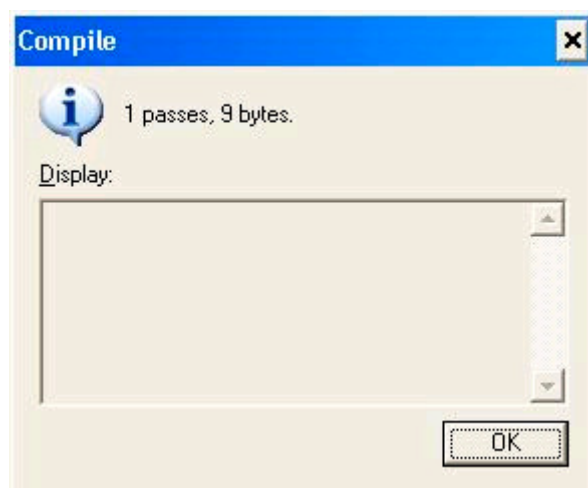
INT 16H
MOV AH, 02H
MOV DL, AL
INT 21H
RET
```

Below the text area is a status bar with a tab labeled 'C.ASM' and a value '6,8'.

Ai está nosso programa, só falta 'MONTA-LO' em um 'Arquivo.COM' da seguinte maneira:



De um nome ao Programa, por exemplo: 'LECHAR' e salve-o, automaticamente deverá aparecer a seguinte caixa de mensagem:



Entre na pasta do FASMW.EXE e lá você poderá executar o 'LECHAR.COM'!

No caso do programa acima ambos os programas funcionarão em um computador IBM PC Compatível com o DOS instalado, sendo assim mais uma vez digo que essa história de compatibilidade, e dificuldades do Assembly é lenda. E se você puder de uma olhada em códigos fontes de jogos como: WolfStein 3D, Doom, Quake e cia, verá que todos esses jogos possuem Assembly IN\_LINE para otimização do jogo, e se você for purista procure pelo jogo chamado: RollerCoaster Tycoon desenvolvido inteiramente em Assembly.

Histórias a parte Finalizo aqui: 'A Primeira Parte do Tutorial Assembly'. Mas não se desespere pois já estou escrevendo a segunda parte. Ainda falta muita coisa a serem vistas como: Loop, Bit a Bit, Lógica Boolean, Segmentação, Dados, Estruturas Avançadas, Otimização e etc.

Por hora fique com a frase que li em um famoso livro sobre Assembly:

'Não Deixe A Lenda Pegar Você!'

**2005 - STIGMA29A - Usuário: Matheus | MSN: matheus\_nab@hotmail.com.**