Assembly para PC

Assembly prático para arquitetura PC (Intel 80XXX)

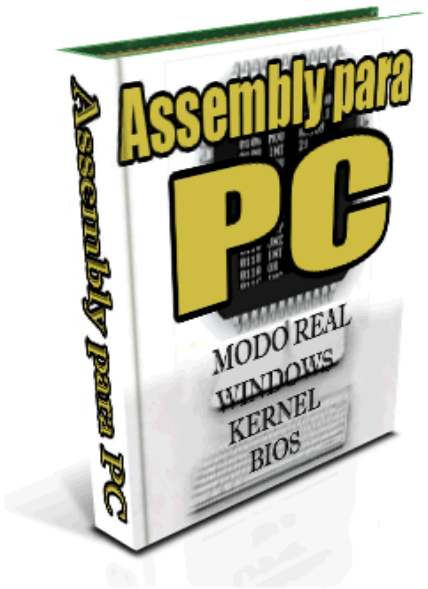
Autor

Dorival Afonso Cardozo

Programador Assembly Intel desde 1987

27 anos programando com Assembly **8 Bits, 16 Bits, 32 Bits e 64 Bits**

Desenvolvendo projetos em **8080,8086,8088,80386**



Curso de **Assembly para PC, 16 Bits, 32 Bits, 64 Bits**. Para Modo Real, Windows, Kernel. Fase Avançada: **Disassembly** e Estudo de **Vírus de Computador**.

Este tutorial é extremamente prático ! você vai aprender como escrever seus primeiros programas em **Assembly** (**Linguagem de Máquina**) em ambiente**Windows e Modo Real**.

Existe também vídeos após cada seção mostrando como executar o que foi ensinado de forma prática (ao vivo), reserve todo dia um tempo para ler este tutorial e ir praticando.
Você vai perceber que este tutorial não é sequencial, ou seja ... o velho método de apresentar instrução a instrução e ir explicando .. não ! não é assim ! prefiro explicar os fundamentos e praticar com instruções simples, para que o aprendizado possa ser intuitivo e escalar.

INTRODUÇÂO

Curso de Assembly para PC

Este curso online é um primeiro passo para entender a linguagem básica do PC que utiliza Intel (**80xx**), depois deste curso espero que todos possam fazer programas básicos e acompanhar a evolução do microprocessador **Intel** que está junto com o PC e o **Windows/DOS** a tantos anos.

Compilador NASM32, FASM
Este curso será progressivo, então vou postar uma matéria eventualmente, juntamente com o vídeo prático de como fazer o que foi aprendido na prática, o compilador será o **MASM32** e o **FASM** (**Flat Assembly**), os 2 compiladores são bons, o primeiro (**MASM32**) não tem recurso 64 bits, mas conhecendo-o, é fácil passar a usar o **FASM** (com recrurso **64 bits**), já que o básico da linguagem é o que interessa, e o **MASM32** tem bastante exemplos, e isto é ótimo.
NASM
Sim, existe um outro chamado **NASM**, que é bem popular no Linux, mas todos compiladores fazem o mesmo serviço, e fazem muitobem, que é de montar o código, e transformar em binário para o processador executar.
O motivo pelo qual escolho o **MASM32** para 32 bits é muito simples ! ele tem o maior arsenal de exemplos (que vi até agora) e é bem simples de entender e usar, e tudo que se precisa quando esta começando é entender as coisas e não sofrer tentando entender explicações.

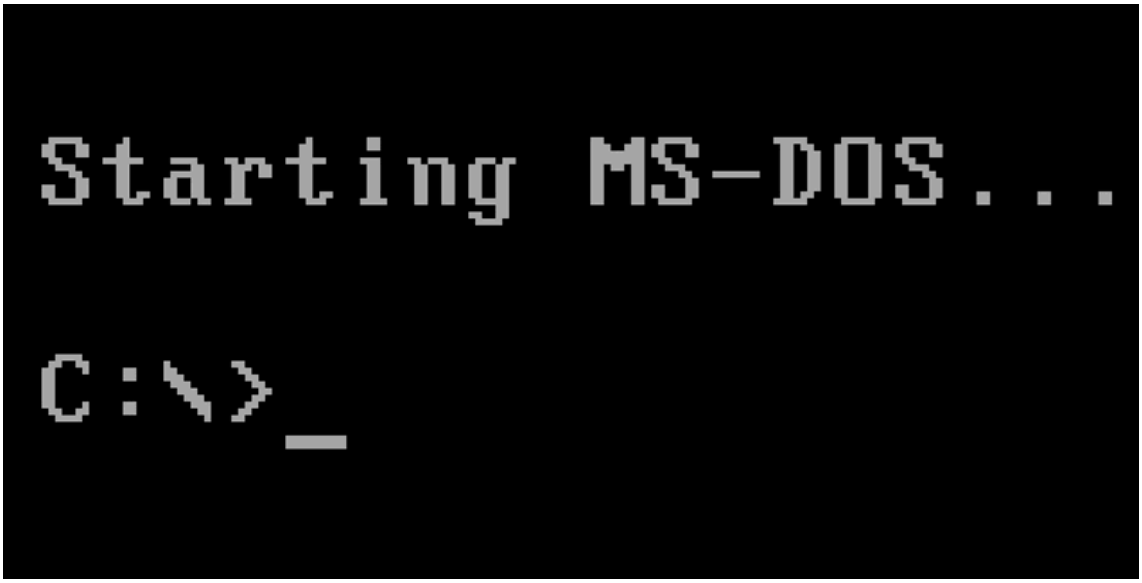
8080/8086/8088/80XX

A história da intel começa a décadas atras, de fato ela passou a frente de suas principais concorrentes que é a Motorola e a Zilog, a motoroloa sempre foi, desde os anos 80 o processador mais bem falado do mercado, esquentava menos, era mais barato,etc ... mas por uma destes ironias do destino, não foi parar no coração do PC. E o motivo, óbvio que foi comercial, quando Bill Gates resolveu fechar acordo com a IBM para fornecer o D.O.S. (Disk Operational System), tentou comprar o CP/M da DIgital Research de Gary Kildal ... mas .. Kildall sequer atendeu a equipe da Microsoft, que na época era um bando de rapazes com cara de nerd, de uma empresa minúscula chamada Microsoft .... depois de dar com cara na porta da Digital Research (que era maior que a microsoft), procurou outro sistema, o que se torno o MS-DOS, desta vez eles foram atendidos e pagaram 50 mi dolares pelo sistema que depois foi melhorado e se transformou no PC-DOS da ibm, e vendido pela microsoft com o nome de MS-DOS, começava aí a fortuna de Gates.

E o que tudo isto tem a ver com Microprocessador e Assembly ? simples ! aquele sistema rodava no Intel 8086, era todo baseado em interrupções de DOS e BIOS, o que hoje no WIndows chamamos de IRP , que são chamados do kernel do WIndows para teclado, mouse, disco, etc ... no ambiente DOS chamava-se “Interrupção de teclado ou Interrupção de disco,etc “.

Então o sistema baseado em Intel foi parar no coração do IBM-PC que depois foi copiado a exaustão no mundo todo, e chegou no Brasil aos milhões via Paraguay, quem se lembra da época, sabe que 90% dos PCs

eram copias do original, que chegavam via Ponte da Amizade no Paraguay e custava 20% do original .. ahh . eo Sistema Operaiconal, lógico, totalmente pirata também, que era o MS-DOS ou PC-DOS.



E acabou assim o reinaldo do Z-80 Zilog , que foi o procesasdor usado na Apolo 11, o foguete que chegou a lua, tudo foi programado em Assembly Z-80 da empresa Zilog em 1969, alias .. a Nasa libero o código fonte em assembly, basta procurar na Internet, tudo em Assembly, o pouso, a estabilidade , e a decolagem ... tudo em baixo nível Zilog, mas não foi suficiente par adominar o mercado de PCs.

A Apolo11 lançada para pousar na lua era toa controlada com Assembly Z-80 Zilog, cuja instruções são bem semelhantes a Intel do PC.

Já a Motorola impera na industria, e muita gente dizia que a qualquer momento os PCs rodariam Motorola e não Intel, ledo engano, isto vem sendo dito a décadas, da mesma forma que dizem que a linguagem Cobol Morreu ... poisé !

E foi assim, sendo o coração do sistema da Microsoft e IBM, a Intel dominou o mundo e depois lançou seu Intel Pentil , um versão superior ao barramento nos anos 80, e por enquanto, é o assembly mai susado no mundos dos PC. Porém, quem der uma olhada nas isntruções assembly da Zilog, vai perceber o quanto ela é semelhante aos registradores da Intel, e talvez seja esta , parte da razão do imperio da Intel.

REGISTRADORES

Vamos começar explicando registradores **16 Bits** (ex. **AX, BX**, etc) mesmo porque os registradores de 32 e 64 bits somente aumentam o tamanho (óbvio !), então sabendo eles, basta “bater o olho” para saber como usar 32 ou 64 bits.

PC ? Sim, vamos começar baseando os exemplo no **PC/WINDOWS/DOS** , futuramente partiremos para o mundo **LINUX**, não muda tanto, e esta é a vantagem de saber baixo nível, o que vai mudar entre **DOS/WINDOWS** e ***LINUX***são as chamados do sistema para realizar alguma tarefa, as operações de movimento de valores, **stack, segmentação de memória, manipulação de registradores** são inerentes ao microprocessador Intel,e portanto são iguais para qualquer sistema operacional.

Juntamente com cada postagem, vou procurar Links que possam adicionar conhecimento ao assunto, aliás, percebi que muita coisa está em inglês, então vou aproveitar para ajudar o **Wikipedia** traduzindo páginas sobre Baixo Nível (**Assembly**) para Português para que exista também uma versão em língua portuguesa.

Para não complicar, e começar pelo fim, é bom ter o **WindowsXP** instalado para começarmos a praticar ... depois passamos para tópicos avançados utilizando**Windows7/Windows8** , o **WindowsXP** é util , porque podemos utilizar o**DEBUG.EXE** que ainda está contido nele, assim como manipulação de áreas que não estarão sob supervisão do sistema, coisa que começou a ficar bem rígida a partir do **Windows vist**a, os **Windows** posteriores já não possuem o **DEBUG.EXE**, ele será útil para praticarmos exemplos simples, este é mais um motivo para ter o**Windows XP** rodando.

Logo no decorrer do curso, vamos utilizando um **Debugger** mais avançado como o**WinDBG,** e até como fazer alguns programas para anexar ao **Kernel** do Windows. Outra coisa, sim ! sou fã do **Linux**, e muito mais que o **Windows**, mas como a maioria dos usuários utilizam Windows, então o **windows** será mais utilizado.

Como funcionam os Registradores Assembly

O registradores armazenam valores, assim como as variáveis que utilizamos em nossa linguagem de programação como (**int valor = 5**) em assembly ficaria (**MOV AX, 5**). No entanto, não podemos sair criando variáveis como em linguagem de alto nível como C++, existe um limite de registradores para colocar valores, assim como outros especiais que indicam offset e segmentos de memoria como (**CS, DS** por exemplo), eles indicam que segmento de memoria o programa esta rodando, e ainda o **IP** que indica que posição de memória está em execução no momento.

Em 32 bits, utilizaremos EAX no lugar de AX, mas aprendendo como utilziar os registradores de 16 bis (AX,BX,CS,etc), fica fácil entender quando eles se transformam em 32 bits (EAX, EBX,ECX), e até quando se transforma 64 bits.

AX.CX, DX, BX, SP, BP, SI, DI

Embora possamo atribuir a eles qualquer valor, alguns dele tem utilidade para o sistema, por exemplo, o SP (stack pointer) é utilizado guardar o valor no Stack, o que é Stack ? é uma área da memória onde são gravado valores via PUSH, POP, CALL por exemplo, (veremos isto mais tarde)
Ainda existem mais registradores de segmento:

CS, DS, ES, FS, GS, SS
Assim como os registradores descritos anteriormente, alguns deles são utilizados pelo ssitema como por exemplo o principal deles (**CS** Code Segment), ele vai ter o valor exatamente do segmento da memória que o programa atual está rodando (também veremos isto no decorrer do curso).
Existe também registradores extras chamados MMX que foi adicionado em meados de 1997, e seu poder é fabuloso ! ele permite manipulação de memoria gráfica de uma forma tão rápida que pode ser cofundida com uma gravação analógica, comentaremos sobre seu poder durante o curso também.

Até agora foi só teoria ... sim, isto pode parecer um pouco chato, mas é essencial para que possamos compreender como “a banda toca” neste bundo maluco do microprocessador, que é na verdade, o coração do processamento de qualquer computador.

Não sera explicado todos fundamentos num capítulo e depois a parte prática, prefiro ir colocando experiência prática e ir explicando , afinal, é assim que aprendemos a andar e a falar esta língua complicada que é o português: Na prática !

Vamos começar com um exemplo simples, exibir uma frase no vídeo, para isto não iremos compilar nada ! vamos inserir diretamente na memória e ver "in natura" como a coisa acontece.

Para isto precisaremos do programa **DEBUG.EXE** que viveu até o **WindowsXP**, então, você deve ter o **WindowsXP** rodando provavelmente em uma máquina virtual como **VirtualBOX** ou **VMWare**, ou pode ainda copiar o **DEBUG.EXE**para o **Windows7** (ou superior) e utiliza-lo para praticarmos.

PRÁTICA

Vamos sair para o sub-mundo do sistema operacional, o mundo **D.O.S.**, para construirmos nosso "**HELLO WORLD**", para isto, estando no windows,

E por que **D.O.S. (prompt de comando)** ? o problema todo é o seguinte, assembly puro só pode ser construido em **D.O.S.**, nesta primeria fase do curso iremos tratar do assembly puro e depois **assembly para Windows** (que exige**API** para funcionar).
Para entender **Assembly**, vamos esquecer as exigências insanas do Windows na construção do código.

um programa que vamos usar muito nesta primeira fase é o DEBUG.EXE, que não está presente no Windows7, Windows Vista nem WIndows 8, parece que o windows vai nos afastando cada vez mais da raiz das coisas ... e consequentemente do Assembly.

Teremos que montar uma plataforma de aprendizado, para isto precisamos de um Windows que pelo menos não fique proibido-nos de fazer as coisas em baixo nível ! o último Windows que foi bonzinho foi

o WIndows XP ! Sim, ele “ainda” tinha o DEBUG.EXE que iremos usar nesta primeira versão, e não também não exigir assinatura de driver, e vamos praticar neste então.

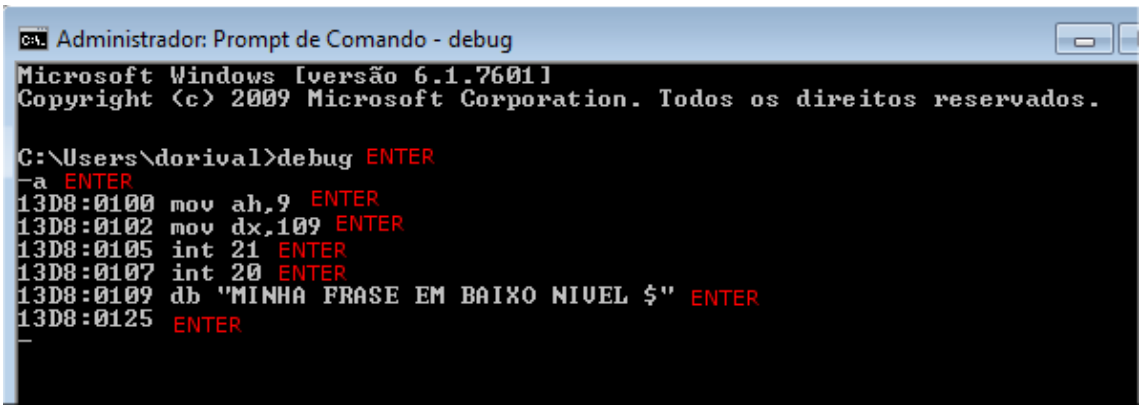
Uma boa técnica é a seguinte, baixar o **VirtualBox** e depois instalar o**WIndowsXP** neste **VirtualBox** (ou **VMWare**), desta forma, conseguiremos executar o **Windows XP** dentro do **Windows7** ou **Windows 8.**

- **Download VirtualBox (se você ainda não tem).**

Depois que instalar o VirtualBox e instalar o WIndows XP , executaremos o Windows XP e seguiremos os passos abaixo.

Entrando em modo D.O.S.

Clicar em **Iniciar->Prompt de Comando ,**
Então rodar o **DEBUG**:



No exemplo acima, após digitar **DEBUG** (e enter para passar a linha de baixo), então digitamos a (enter) e digitamos as instruções **assembly** descritas acima, espero que esteja claro que este

ENTER vermelho se refira a apertar **ENTER** (ou apertar **RETURN**) para encerrar a linha, eu não vou mais colocar este **ENTER** em vermelho nos próximos exemplos.

Video YouTube sobre este exemplo

https://www.youtube.com/watch?v=rhnGz2bi608

Comentando linha a linha:

"-a " Este **(a)** digitado dentro do **DEBUG**, informa ao **DEBUG** para acessar a próxima memória disponível, que seria **100h**, antigamente todos programas .COM começavam em **100h**, então ele já assume este endereço como padrão, como não vamos fazer programas grandes **.EXE,** podemos utilizar este endereço por enquanto.

"MOV AH,9 " MOV é uma instrução que informa ao microprocessador para mover o valor 9 para a parte alta do registrador AX, e o que seria isto ?
Vamos entender: O Registrador **AX** possui **16 bits** ! ou seja, ele também pode ser dividido em 2 partes, podemos imaginá-lo como aquelas células que se dividem e possuem vida própria, ou seja, este AX pode ser dividido em dois registradores de 8 bits chamado AH e AL (AH = parte alta, AL= parte baixa) (**H=High, L=Lo**w), antigamente nos idos **anos 80** (Bee Gees, ABBA ... isto não é do seu tempo né ?) isto era tudo que existia nos antigos computadores de 8 bits que andaram pelo mundo como primeiro microcomputadores baseados em 8080 ou Z80, eles tinham somente 8 bits (metade do **AX**). Logo no comecinho dos anos 90 os computadores de 17 bits ficaram mais baratinhos, e invadiram o mundo todo, então o registrador**AX** começou a ser utilizado também, embora o recurso de utilizar metade dele (AH ou AL) ainda é usado para diversos fins, inclusive o fim deste programa. Este texto não acabou ! amos falar mais sobre esta linha, o que significa este 9 colocado no AH ? ele informa o sistema simplesmente para exibir uma frase quando for chamado a interrupção **INT 21h** logo na frente.

"MOV DX,109" Conforme se leu no texto anterior, o valor 9 em AH informa o sistema que uma frase será impressa quando se chamar o

INT 21 logo a frente (ele só vai exibir quando a interrupção **INT 21h** for chamada), mas ... onde fica o tal texto que será exibido ? é este o objetivo desta linha, indicar que local da memória esta a frase, que seria o endereço 109, inserindo o número 109 em DX, já informamos o sistema onde fica o tal texto afinal, o texto em questão é "MINHA FRASE EM BAIXO NIVEL $" no endereço de memória **109**.

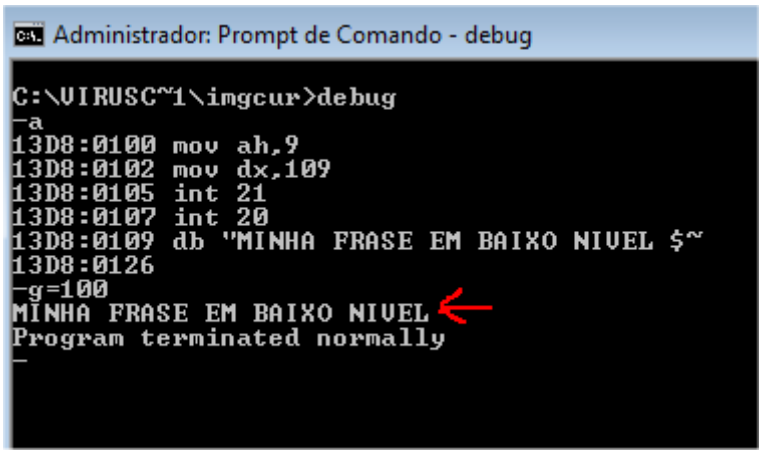
"INT 21" Este é o cara ! só quando esta interrupção é chamada a frase é exibida, então nada adianta colocar os valore sem DX, em AH se esta interrupção não for chamada, quando ela é acionada o sistema vai pegar seu trabalho em AH (é onde ele fica sabendo o que fazer) e depois sabendo que é para exibir um texto, vai pegar o texto em DX que adivinha ? tem o valor 109 que é o endereço do texto.

"db "MINHA FRASE A SER EXIBIDA" Perceba o número que aparece anes de **db**, é o endereço de memória **109**, então o db (data byte) somente ifnorma ao DEBUG para inserir o texto a frente byte a byte começando no endereço **109**, no final do texto vemos o dolar (**$**) e porque ele está no final ? ele só i vai informar o fim do texto, se não for inserido a interrrupção INT 21 vai imprimindo tudo que estiver na frente do texto sem parar até o fim da memória ou até encontrar um **$** no caminho, ele delimita o fim mas não é impresso.
"INT 20" Esta interrupção somente informa que o programa terminou, sim ! em assembly precisamos informar que nossa festinha acabou, ela pode ser comparada ao " } " do C ou do "end." do pascal (ou delphi).

Bem, até agora só vimos teoria, não vimos nada pular na tela, nada acontecer ... vamos fazer algo acontecer, vamos RODAR o programa, fazer ele exibir a tal frase, depois vamos gerar um programa com ele SEM COMPILAR NADA ! como ? você se esqueceu que assembly é a inlguagem do microprocessador ? e estamos escrevendo diretamente para ele ? só compilamos quando escrevemos em linguagem de alto nivel como **C , Pascal,** etc., e quando compilamos o que acontece ? se transforma em **assembly**.

Para executar, basta digitar G=endereço, onde o endereço é o ... 100h , então ficaria (g = 100).

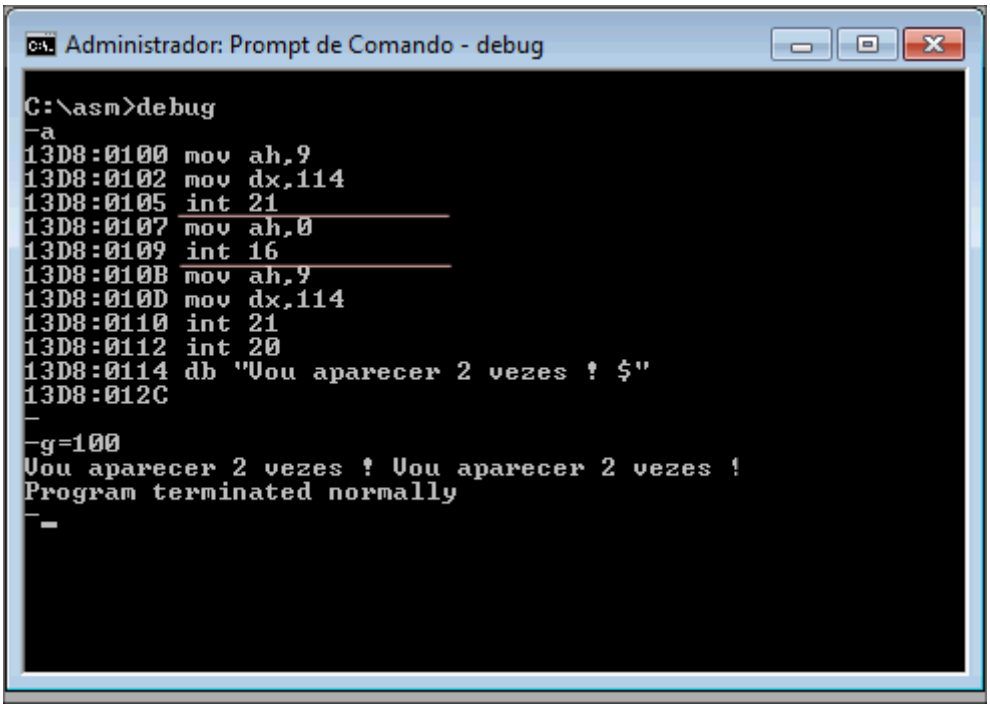
Como visto abaixo:



Entendendo melhor este programinha aí, poderíamos descreve-lo em algoritmo da seguinte forma:

***AH* = *9*oque faz**: 9 Informa o sistema que algo será exibido
***DX* = *endereco* oque faz**: DX = endereço, DX sempre vai ter o endereço de memória do que desejamos exibir
***Chamar Interrupção 21h*oque faz:** Executa a operação descrita em AH (que é 9, exibição de texto, mas poderia ser outro processamento)
Chamar Interrupção 20hoque faz: Termina o programa, sim ! a festa termina aqui

Vamos continuar explorando mais os recursos do **DEBUG**, que tal esperar uma tecla antes de terminar o programa ? o famoso **INPUT** ! Vamos utilizar uma interrupção diferente da velha **21h**, vamos utilizar a interrupção responsável pelo teclado, a interrupção **16h ,** e dando uma olhadinha de como utilizar, vemos que precisamos inserir o valor **0 em AH**.



No exemplo acima, saímos da mesmice da impressão de texto, e lemos uma tecla, para começar vemos o primeiro bloco com ah=9, dx=114 e o velho int 21 para exibir o texto, logo após vemos algo diferente, AH=0 e INT 16, que não exibe nada, apenas espera que uma tecla seja digitada, após pressionamos qualquer tecla, novamente exibimos a mesma frase.

Para executar o programa, vemos lá o G=100 (comece a executar a partir do endereço 100h, perceba que começa realmente em 100, olhe lá o endereço 13D8:0100h , ignore este 13D8 , este é o segmento de memória (explicarei mais tarde), e o segmento muda constantemente, o que interessa é o OFFSET 100h mesmo).

Logo abaixo do G=100, vemos o texto exibido, aguarda uma tecla ser pressionada, e novamente exibido,e depois a mensagem "**Program terminated normally**".

E o que significa tudo isto ? só prática mesmo ! para irmos nos acostumando a usar o MOV para inserir valor em um registrador, e ver

como diferentes interrupções fazem coisas diferentes, prática é tudo para sair fazendo sem precisar pensar, assim como fazemos quando precisamos aprender uma língua nova ... só praticando vamos ficando fluente nisto,e a instrução MOV é a mais básica de todas, assim como é utilizada para mover valores em registrador 8 bits (**MOV AH, 9**), também é utilizada para 16 bits (**MOV AX, 100h**) e 32 bits (**MOV EAX, 100h**).
Espero ter convencido que isto que foi apreendido é muito útil didaticamente.

GERANDO UM PROGRAMA EXECUTÁVEL SEM COMPILAR NADA !

Que tal gerar um programa ? até agora inserimos instruções na memória, depois desligamos o PC e perdemos tudo ! ora ... como manter isto em um programa ? vamos então gerar através do **DEBUG** mesmo.
Mas antes ! vamos entender como isto funciona (lá vem teoria !) :
Vamos entender como funciona a evolução de alguns executáveis do Windows, pelo menos os principais:

Programas **.COM** (APPLE.COM)
Programas **.EXE Comum** (APPLE.EXE)
Programass **.EXE Windows** (APPLE.EXE)

Existem outros tipos de executáveis, mas vamos nos ater a estes que marcaram a evolução dos executáveis, o primeiro é este **.COM ,** este é o inicio de tudo ! Quando Bill Gates comprou o **D.O.S.** para vender a IBM como se fosse dele , já era assim ! o sistema rodava nos **640k básicos**, e tudo era feito ali, o sistema de contabilidade da empresa e até a folha de pagamento.
Então os programas .COM não poderiam ser maior que 1 segmento de memória, segmento é aquela memória que aparece a esqueda

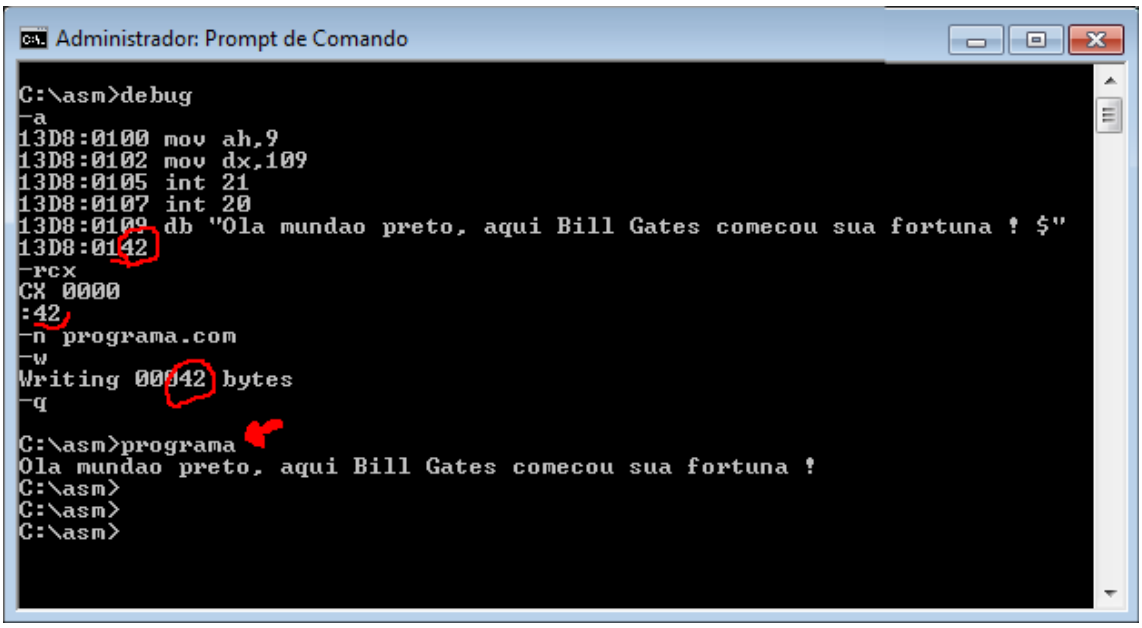
(**segmento : offset**). Como vimos no exemplo acima, o valor do segmento é sempre o mesmo, o que muda é o valor do offset que é a extensão do segmento, podemos imaginar uma matriz também.

Com o tempo as pessoas começaram a precisar de mais memória, então surgiu o**.EXE** , ele podia armazenar o programas além de um

segmento, isto foi formidável para fazer **Vírus de Computador**, porque o vírus podia escolher um segmento para ele mesmo rodar, e colocar a vítima em outro segmento, e isto organizava a programação para o **Vírus Maker**, isto ainda é utilizado hoje nos Malware para Windows, esta manipulação de segmentação que facilita a compreensão das coisas.
Os programas **EXE** são bem diferentes dos programas .COM, porque eles tem um cabeçalho para avisar os sistema como alocar o programa, onde ele começa, onde ficam as seções de dados e imagens do programas, isto não existe nos programas .COM, do inicio ao fim é o programas em si ! enquanto em tipos **.EXE** existe um cabeçalho que não faz parte das coisas que programamos.
Daí vem os **.EXE** para Windows, eles possuem ainda mais um cabeçalho para informar o Windows como seu programas será alojado na memória, etc.

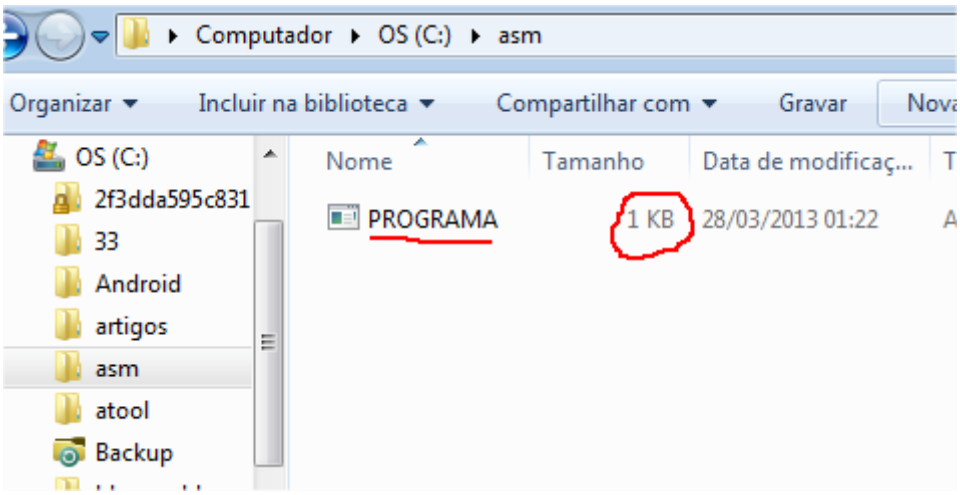
No futuro iremos gerar programas .EXE em Assembly, e veremos ele dissecado em nossa frente.

No momento vamos gerar um programa **.COM** , e por ser um tipo tão simples, não vamos ter dor de cabeça.



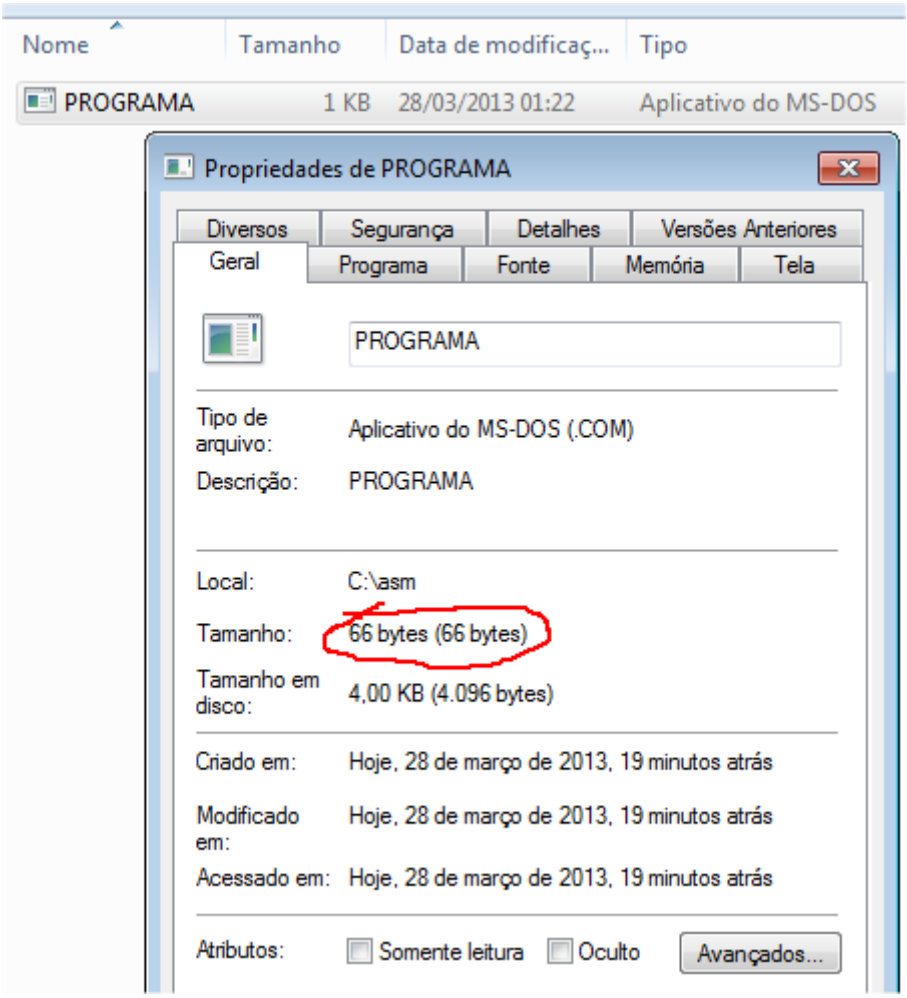
No exemplo acima, criamos um arquivo chamado PROGRAMA.EXE, vamos explicar o que fizemos ali encima:

Logo apos digitarmos o programa, isto já fizemos nos exemplos anteriores, no endereço 142 digitamos ENTER sem digitar nada, então o DEBUG sai no modo de edição de memória, daí utilizamos a diretiva **rcx** , ela é utilizada para informar que tamanho terá nosso programa, então colocamos o tamanho 42h (sim ! todo número que vemos dentro do debg é HEXA !, portanto o número **42h** significa **66 bytes** no sistema decimal que conhecemos).
Depois digitamos **n** programa.com (**n = nome do programa**), colocando programa.com a frente do " **n** ", informamos que nome vamos utilizar.
O **w** informa o **debug** para gravar o programa que digitamos até o tamanho **42h** , utilizando o nome **programa.com**.
Logo depois, digitamos o nome do programa e vemos ele exibir a frase "**Ola mundao preto, aqu Bill Gates comecou sua fortuna !**".
Voialá ! Este programa não perdemos mais ! está gravado e até podemos ver através do Windows Explorer.



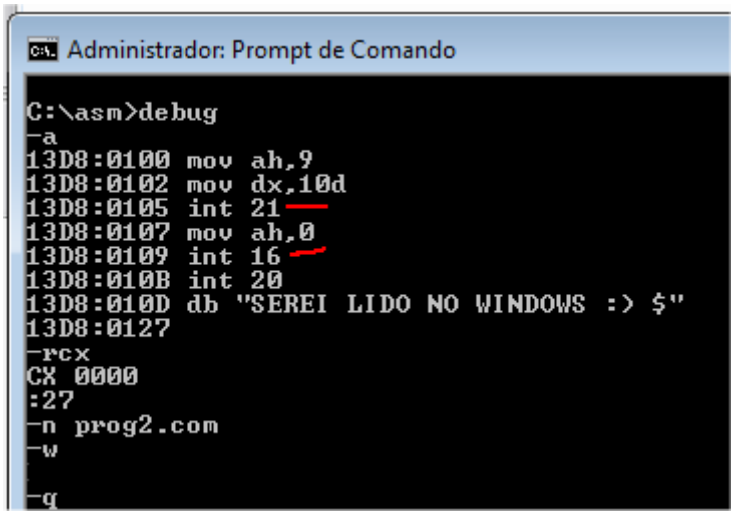
Através do **Windows Explorer**, vemos o programa **PROGRAMA.COM** criado, no tamanho veremos o tamanho **1Kb**, que na verdade é falso ! porque o tamanho gravado foi de 66 bytes (42 hexa lembra ?), é estupidamente pequeno ! absurdamente pequeno ! que compilador gera um programa no tamanho de 66 bytes ? mas o windows não consegue mensurar isto,e coloca 1Kb.

Vamos dar uma olhada na propriedades do programa através do Windows mesmo:



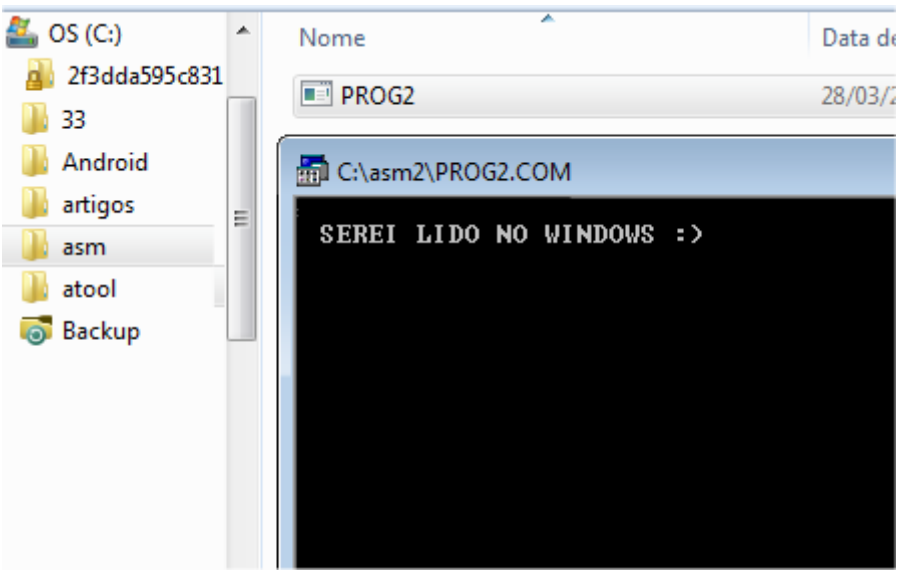
Aí está ! agora o Windows mostrou certinho o tamanho absurdamente pequeno de nosso programa.com, isto ocorre porque o executável não carrega nenhuma tabela de alocação e é totalmente inserido em **Assembly** sem passar pro nenhum compilador.

Bem, agora vamos rodar em Windows, veremos ele abrir uma tela e fechar, e isto é ... um pouco decepcionante, porque a frase aparece tão rapidamente, que não vemos ela, que tal pedir para o programa esperar um ENTER antes de terminar ? ahhh podemos ver a frase então, vamos lá:



Aqui vemos nosso programa que exibe uma frase e... espera por uma tecla apertada.

Rodando ele no Windows, vemos então a frase já que vai esperar algo ser teclado.



Mais Instruções Assembly

Vamos aprender novas instruções e como utiliza-las , na verdade não será aplicado todas as instruções, mas as principais; para aprender

precisamos saber o básico, mesmo o básico já exige bastante, e também ficaremos um pouquinho aqui no ambiente do DEBUG.EXE para aprendermos as instruções fundamentais.

Da mesma forma, quando aprendemos a dirigir um carro sempre usamos um carrinho simples e popular, e também fazemos as aulas básicas e fundamentais, suficiente para dirigirmos mais tarde um BMW igual do Charlie Harper (two and half man), da mesma forma, estaremos aptos a dirigir em qualquer trânsito do mundo, sem precisar fazer aulas para dirigir no mundo todo para isto, é óbvio. Ficaremos aqui no "fusquinha" do ambiente assembly para aprender o fundamental, pelo menos no início.

Vamos agora ver como COMPARAR valores, e também como usar um contador para medir quantas vezes se passa por uma rotina.

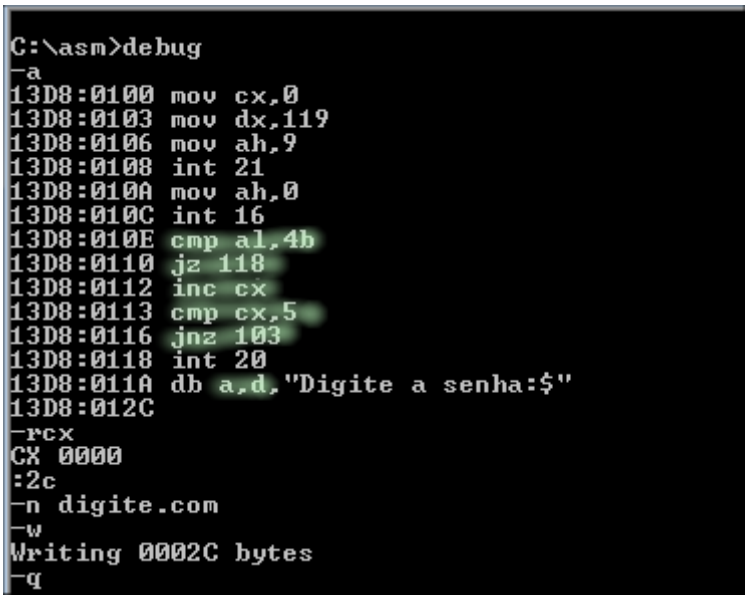
O programa a seguir vai fazer o seguinte:

*1- Exibir a frase **"Digite a Senha"***
2- Esoerar a senha pelo teclado
*3- Comparar a senha (tamanho de **1 Byte)** , com a letra **K ,** vai ser necessário comparar o código do **K** e não o **K** , seu código é **ASC é 75**, e em hexa **4Bh***
*4- Se for a letra **K**, passo **7** para terminar o programa*
*5- Se não for K, verificar se chegamos a **5a**. tentativa, se chegou pular para **7***
*6 – Voltar ao passo **1** e tentar novamente*
7- Termine o Programa

Já de cara, percebemos que precisamos de um contador, ele é necessário para contar quantas vezes estamos tentando, já que na 5a. vez precisaremos terminar o programa.

Outra coisa que ainda não foi explicado, é a instrução de compração "**CMP**" e a instrução para puilar se a comparação for verdaderia "**JZ endereço**".
Outra coisa nova que pode ser feito enquanto estamos construindo o programa é a gravação em tamanho grande **(R CX=200** por exemplo), como ele é grande, também está mais sujeito a erros de programação, para isto, podemos portanto gravar nosso programa com um tamanho grande, já que não sabemos o tamanho total no final, então quando

estiver testado, ajustamos o tamanho do program (**com R CX=tamanho**).



Aí esta o programa, as instruções novas estão destacadas em verde e são elas:

CMP AL, 4B

Esta instrução compara **AL** (a parte low do **AX**) , compara portanto com **4B** , e o que será **4B** ? é o código hexa de **K ,**
não cofunda, o código ASC de K é **75** , mas lá só vale o valor em hexa, então convertendo **75** em hexa, temos **4Bh**

E de onde aparece este valor em **AL** ? após um **Int 16** (Instrução anteriro), o sistema vai colocar em **AL** o valor digitado,
se for digitado a letra **A**, após o Int 16, **AL** vai valer **41h** que é o código de **A.**

JZ 118

Esta instrução significa (Jump se Zero), ou seja, se a comparação anterior tiver sucesso, ou seja, se AL = 4B, então o Flag será setado em Zero ! então basta comprar com JZ, e se for zero mesmo, ele salta para o endereço 118, perceba que em no endereço 118 tem o INT 20 que é ? fim de programa.

INC CX

Se você pensou em **Incrementation** , acertou ! ele incrementa 1 valor no registrador **CX** , seria como (**CX = CX + 1** ou **CX++** como faríamos em **C**), e porque incrementa 1 ? para ir contando as tentativas, perceba que a instrução abaixo compara com 5.

CMP CX, 5

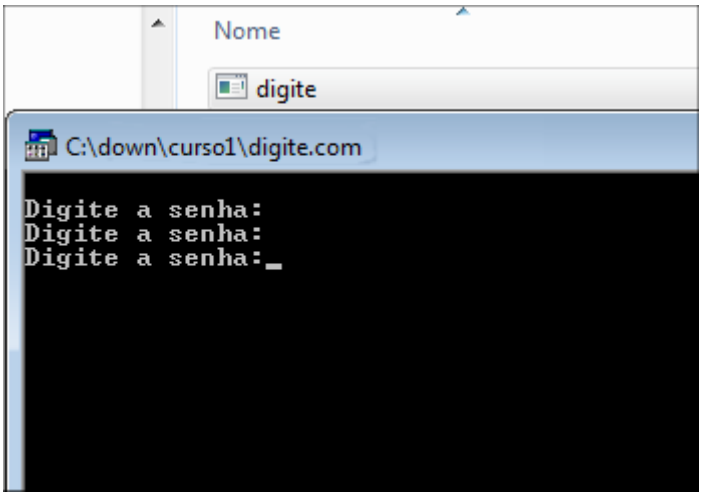
Esta é fácil ! já aprendemos o que faz o CMP, aqui ele compara CX co 5, e se for 5 mesmo, ele salta para o fim do programa

JNZ 103

Esta também sabemos, intuitivamente podemos perceber que se JZ salta para um endereço quando a comparação do CMP obteve sucesso, utilizando JNZ ele salta se NÂO obteve sucesso, neste caso a comparação da instrução anterior (CMP CX, 5) se CX não chegou a 5, ele salta para o inicio do programa e começa tudo de novo.

Observe também que antes da frase tem o código: **a,d,"Digite a senha:"** , este a é o **10 (line feed)**, e o d é o **13 (Return)**, ou seja, enquanto o 10 pula para linha de baixo, o **13** volta para o inicio, é por isto que toda vez que erramos ele pula para linha de baixo e refaz o texto.

No exemplo o programa também é gerado com o nome digite.com e gravado no disco, podemos testa-lo no windows para ver o que acontece.

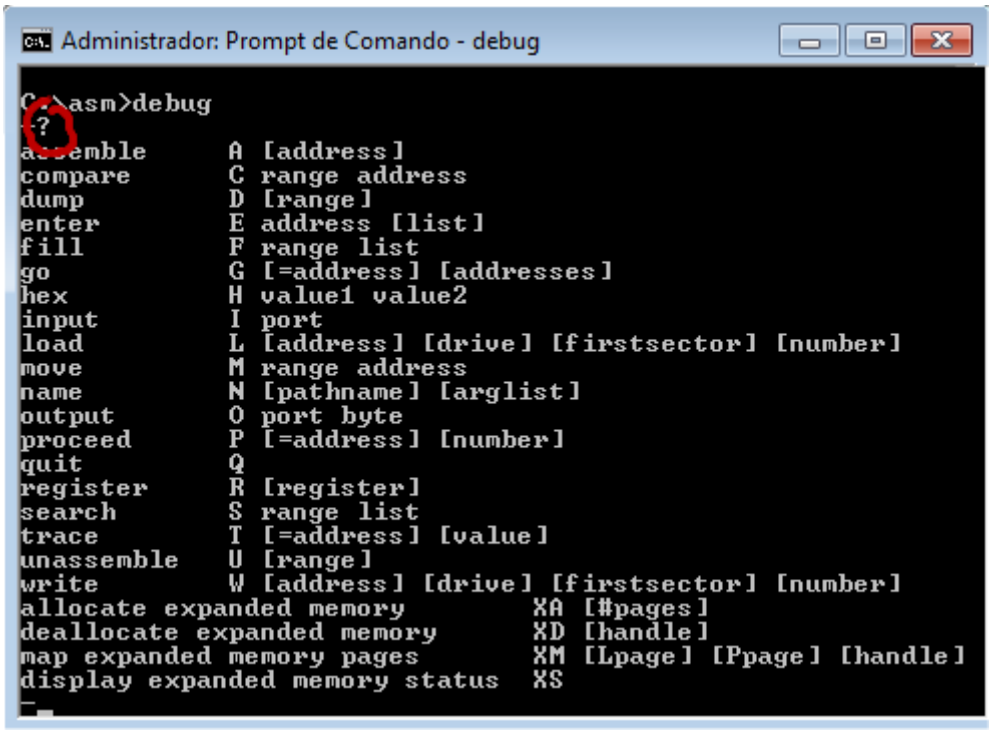


Imagino que este programa começou a assustar, então está aqui uma opção de fazer download deste digite.com.

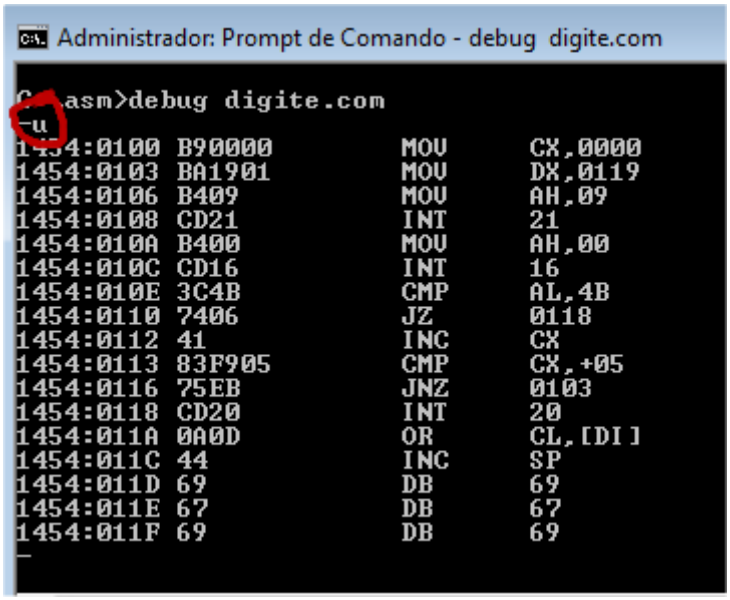
DOWNLOAD DO EXECUTÁVEL:[digite.com]
SE PREFERIR (e seu anti-virus exigir ! **) PEGUE A VERSÃO COMPACTADA:[digite.rar]**

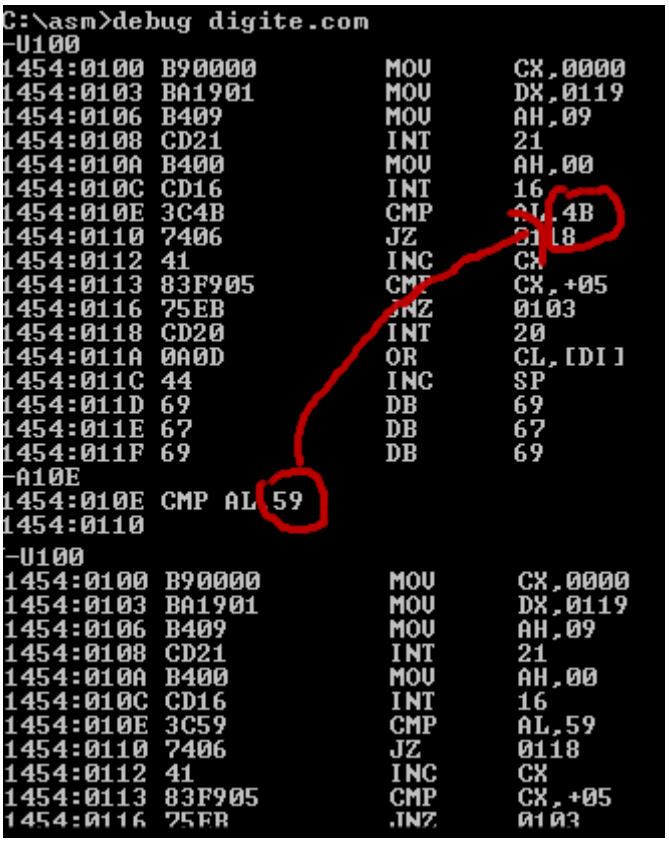
EDITANDO PROGRAMA PRONTO

Como alterar um programa já pronto ? vamos fazer isto utilizando o DEBUG, que tal alterar aquela frase do programa anterror ? e alterar também a senha, mudar de**K** para **Y**.
Primeiramente vamos ao código do **Y** maiúsculo, o código **ASC** é **89 decimal**, então convertemos para Hexa que é **59h.**
Para fazer a edição, basta digitar **DEBUG digite.com** , para se ver o programa, vamos utilizar a diretiva U do **DEBUG** , podemos vê-lo no help do **DEBUG**, para ver este Help pasta digitar **?** no **DEBUG**.

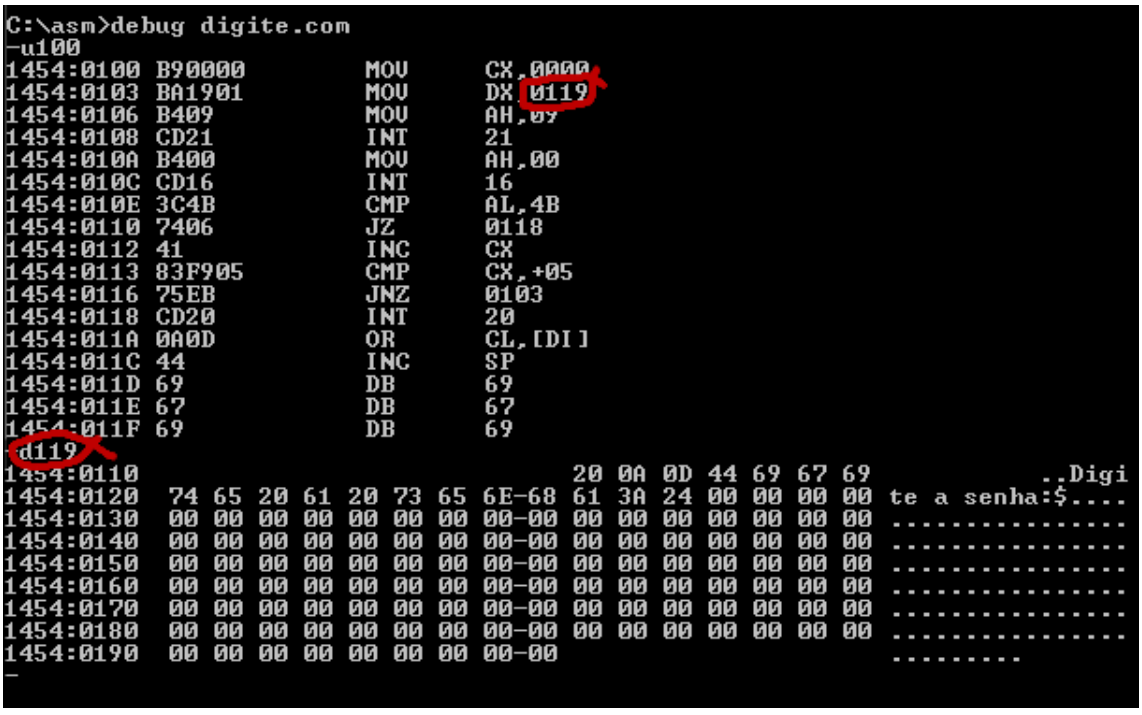


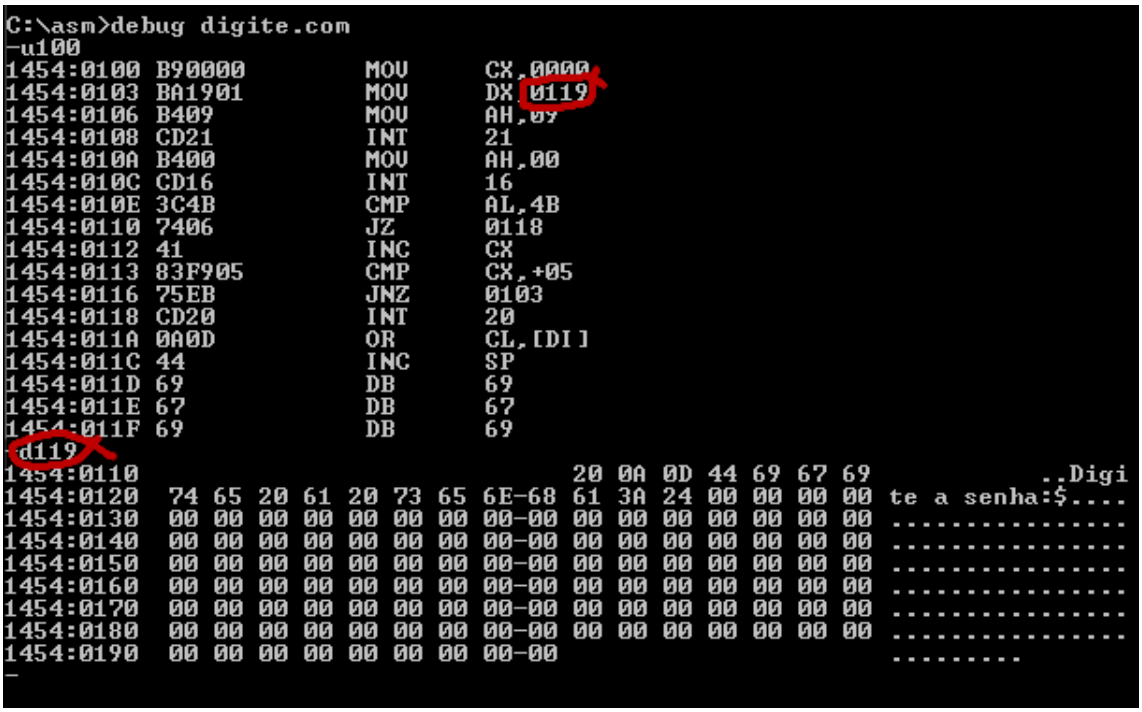
Alterando o código da tecla senha de **K** para **Y,** para isto basta editar o endereço**10E** com a diretiva **"A 10E"** e digitar
a mova instrução **assembly "CMP al, 59".**

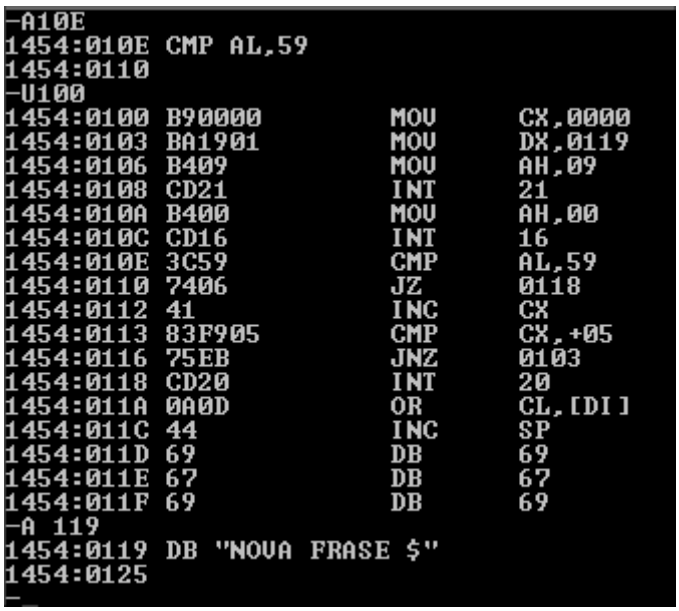




O próximo passo é alterar a frase exibida, para isto basta repetir o último procedimento, porém com endereço diferente, desta vez para alterar o endereço de memória 103:







Neste último exemplo foi demonstrado como alterar também o texto do endereço**119**, bastando digitar **DB “novo texto”.**
Para gravar esta alteração basta seguir os procedimentos que usamos quando criamos, utilizar a diretiva do debug **W** para gravar esta alteração.