MANUAL DO ALUNO

DISCIPLINA SISTEMAS DIGITAIS E ARQUITETURA DE COMPUTADORES

Módulo 9

República Democrática de Timor-Leste
Ministério da Educação

FICHA TÉCNICA

TÍTULO
MANUAL DO ALUNO - DISCIPLINA DE SISTEMAS DIGITAIS E ARQUITETURA DE COMPUTADORES
Módulo 9

AUTOR
BRUNO MORAIS

COLABORAÇÃO DAS EQUIPAS TÉCNICAS TIMORENSES DA DISCIPLINA
XXXXXXX

COLABORAÇÃO TÉCNICA NA REVISÃO

DESIGN E PAGINAÇÃO
UNDESIGN - JOAO PAULO VILHENA
EVOLUA.PT

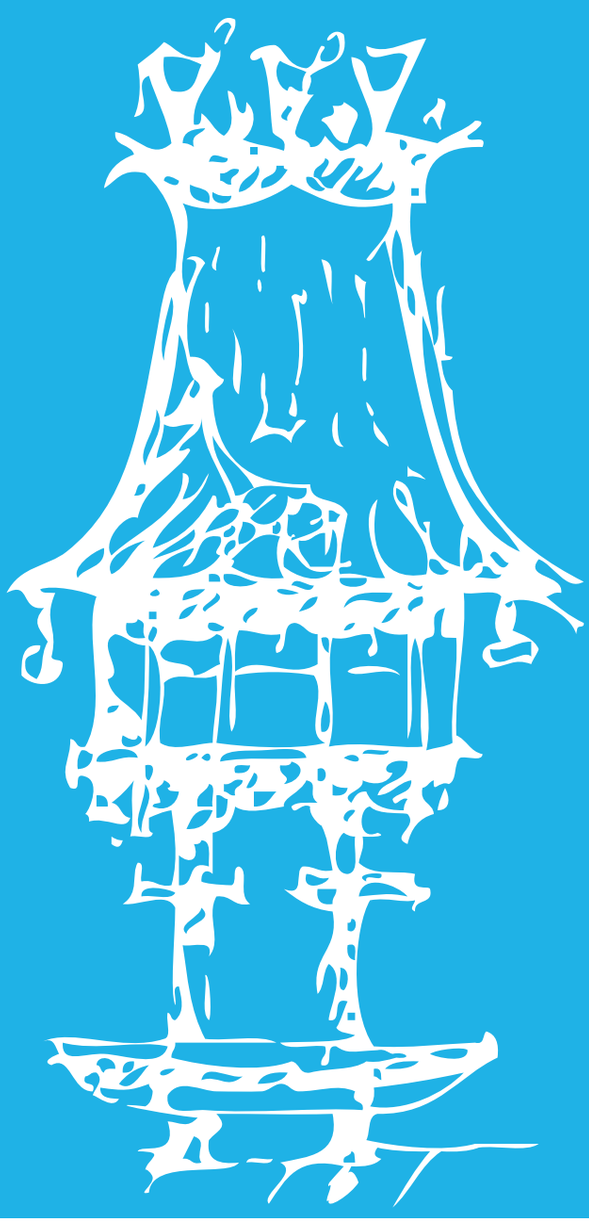
IMPRESSÃO E ACABAMENTO
XXXXXX

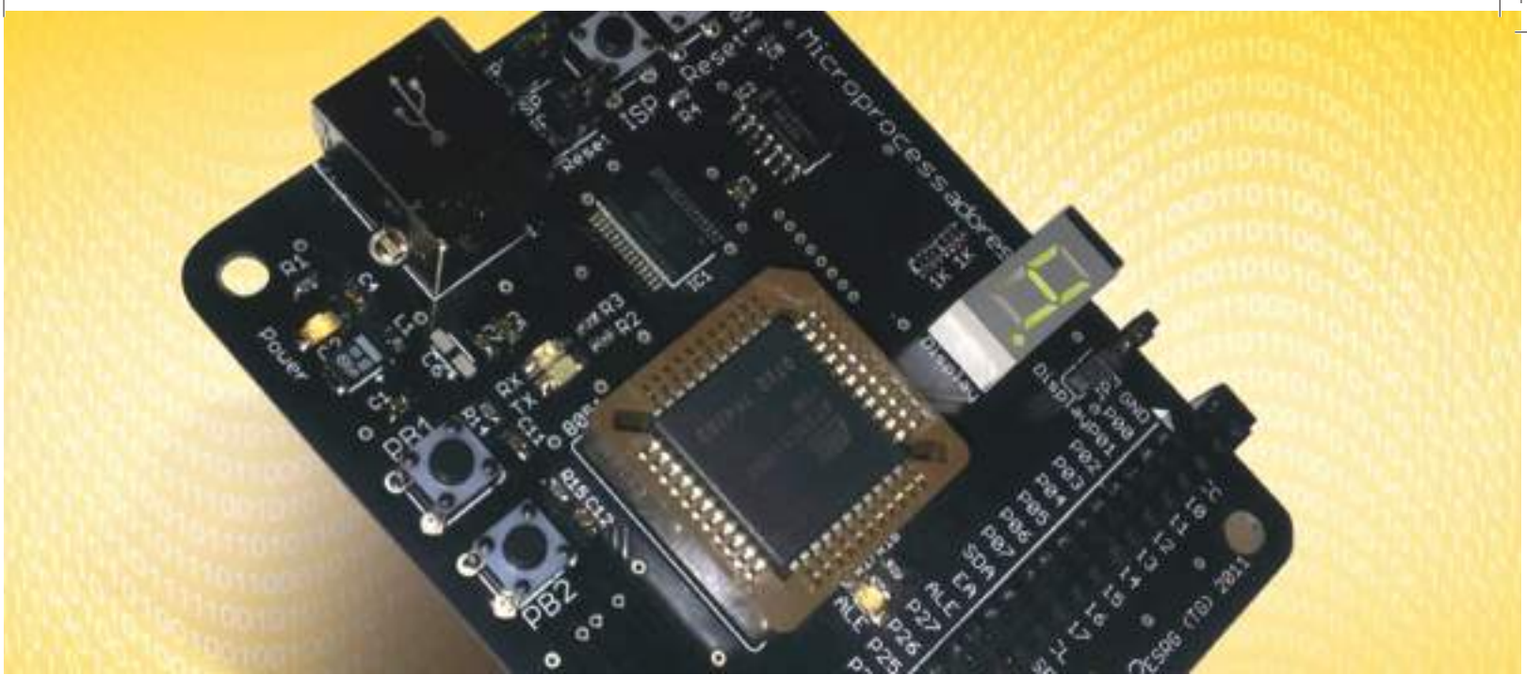
ISBN
XXX - XXX - X - XXXXX - X

TIRAGEM
XXXXXXX EXEMPLARES

COORDENAÇÃO GERAL DO PROJETO
MINISTÉRIO DA EDUCAÇÃO DE TIMOR-LESTE
2015

Índice





Programação de Microprocessadores

Módulo 9

Caracterização do Módulo

Apresentação

Este módulo pretende familiarizar os alunos com as técnicas de desenvolvimento de aplicações baseadas em microprocessadores e microcontroladores quer no que respeita ao desenvolvimento de software, com ênfase na programação modular em assembly, quer na sua relação com o hardware de suporte.

Objetivos de aprendizagem

Dominar os conceitos básicos de programação em *Assembly*.

Realizar "*debugging*" de pequenos programas em *Assembly* utilizando o utilitário apropriado.

Estudar o funcionamento de um Sistema Operativo a baixo nível.

Realizar programas simples de exemplo em *Assembly* de comunicação com o exterior, que utilizem recursos disponíveis pelo Sistema Operativo através de *SYSTEM CALLS* (chamadas ao sistema).

Âmbito de conteúdos

1. *"Set"* das principais instruções de um Microprocessador tipo.
2. Realização de pequenos programas de acesso à memória de vídeo como exemplo de aplicação do *Set* de instruções.
3. Noção de rotinas e principais conceitos a ela associados.
4. Passagem de parâmetros a rotinas por endereço e por valor.
5. Principais conceitos associados à utilização de Interrupções num computador.
6. Utilização dos utilitários disponíveis para fazer "*debugging*".
7. Estrutura interna de um sistema operativo tipo.
8. Noção de chamadas ao sistema.
9. Principais chamadas ao sistema disponíveis por um sistema operativo tipo.

10. Utilização das funções de um S.O., para tratamento de ficheiros (Ex: carregar uma imagem, ou uma música para um *buffer* em memória previamente alocado).
11. Conceito de "*device drivers*".

Linguagem Assembly

Introdução

A primeira razão para se trabalhar com o assembler é a oportunidade de conhecer melhor o funcionamento do seu PC, o que permite o desenvolvimento de programas de forma mais consistente.

A segunda razão é que podemos ter um controlo total sobre o PC ao fazer uso do assembler.

Uma outra razão é que programas assembly são mais rápidos, menores e mais poderosos do que os criados com outras linguagens.

Ultimamente, o assembler (montador) permite uma otimização ideal nos programas, seja no seu tamanho ou execução.

Descrição de um sistema computacional

Chamamos de Sistema Computacional a completa configuração de um computador, incluindo os periféricos e o sistema operacional. Vamos a seguir fazer um breve resumo de matérias já lecionadas para relembrar alguns conceitos.

Processador Central

Como já vimos em módulos anteriores é também conhecido por CPU ou Unidade Central de Processamento, que por sua vez é composta pela unidade de controlo e unidade de lógica e aritmética. A sua função consiste na leitura e escrita do conteúdo das células de memória, regular o tráfego de dados entre as células de memória e registadores especiais, decodificar e executar as instruções de um programa.

O processador tem uma série de células de memória usadas com frequência e, dessa forma, são partes da CPU. Estas células são conhecidas com o nome de registadores. Um processador de um PC possui cerca de 14 registadores.

Como os PCs tem sofrido evolução veremos que podemos manipular registadores de 16 ou 32 bits.

A unidade de lógica e aritmética da CPU realiza as operações relacionadas ao cálculo simbólico e numérico.

Tipicamente estas unidades apenas são capazes de realizar operações elementares, tais como: adição e subtração de dois números inteiros, multiplicação e divisão de número inteiro, manuseamento de bits de registadores e comparação do conteúdo de dois registadores.

Computadores pessoais podem ser classificados pelo que é conhecido como tamanho da palavra, isto é, a quantidade de bits que o processador é capaz de manusear de uma só vez.

Memória Principal

É um grupo de células, que agora são fabricadas com semicondutores, usada para processamentos gerais, tais como a execução de programas e o armazenamento de informações para operações.

Cada uma das células pode conter um valor numérico e é capaz de ser endereçada, isto é, pode ser identificada de forma singular em relação às outras células pelo uso de um número ou endereço.

O nome genérico destas memórias é Random Access Memory ou RAM. A principal desvantagem deste tipo de memória é o fato de que os seus circuitos integrados perdem a informação que armazenavam quando a energia elétrica for interrompida, ou seja, ela é volátil. Este foi o motivo que levou à criação de um outro tipo de memória cuja informação não é perdida quando o sistema é desligado. Estas memórias receberam o nome de Read Only Memory ou ROM.

Unidades de Entrada e Saída

Para que o computador possa ser útil para nós é necessário que o processador se comunique com o exterior através de interfaces que permitem a entrada e a saída de informação entre ele e a memória. Através do uso destas comunicações é possível introduzir informação a ser processada e mais tarde visualizar os dados processados.

Algumas das mais comuns unidades de entrada são o teclado e o rato. As mais comuns unidades de saída são o ecrã do monitor e a impressora.

Unidades de Memória Auxiliar

Considerando o alto custo da memória principal e também o tamanho das aplicações atualmente, vemos que ela é muito limitada. Logo, surgiu a necessidade da criação de dispositivos de armazenamento práticos e económicos.

Estes e outros inconvenientes deram lugar às unidades de memória auxiliar, periféricos. As mais comuns são as pen drivers e os discos rígidos.

A informação ali armazenada será dividida em ficheiros. Um ficheiro é feito de um número variável de registos, geralmente de tamanho fixo, podendo conter informação ou programas.

Conceitos Básicos da Linguagem Assembly

Informações nos computadores

Unidades de informação

Para o PC processar a informação, é necessário que ela esteja em células especiais, chamadas de registadores.

Os registadores são grupos de 8 ou 16 flip-flops.

Um flip-flop é um dispositivo capaz de armazenar 2 níveis de voltagem, um baixo, geralmente 0.5 volts, e outro alto comumente de 5 volts. O nível baixo de energia no flip-flop é interpretado como desligado ou 0, e o nível alto, como ligado ou 1. Estes estados são geralmente conhecidos como bits, que são a menor unidade de informação num computador.

Um grupo de 16 bits é conhecido como palavra; uma palavra pode ser dividida em grupos de 8 bits chamados bytes, e grupos de 4 bits chamados nibbles.

Sistemas numéricos

O sistema numérico que nós usamos diariamente é o decimal, mas este sistema não é conveniente para máquinas, pois ali as informações têm que ser codificadas de modo a interpretar os estados da corrente (ligado-desligado); este tipo de código faz com que tenhamos que conhecer o cálculo posicional que nos permitirá expressar um número em qualquer base onde precisarmos dele.

Conversão de números binários para decimais

Quando trabalhamos com a Linguagem Assembly encontramos por acaso a necessidade de converter números de um sistema binário, que é usado em computadores, para o sistema decimal usado pelas pessoas.

O sistema binário é baseado em apenas duas condições ou estados, estar ligado(1), ou desligado(0), portanto sua base é dois.

Para a conversão, podemos usar a fórmula de valor posicional:

Por exemplo, se tivermos o número binário 10011, usamos cada dígito da direita para a esquerda e multiplicamo-lo pela base, elevando à potência correspondente à sua posição relativa:

Binário: 1 1 0 0 1

Decimal: 1*2^0 + 1*2^1 + 0*2^2 + 0*2^3 + 1*2^4

O caracter ^ é usado em computação como símbolo para potência e * para a multiplicação.

Conversão de números decimais para binário

Há vários métodos para se converter números decimais para binário; apenas um será analisado aqui. Naturalmente a conversão com uma calculadora científica é muito mais fácil, mas nem sempre podemos contar com isso, logo o mais conveniente é, ao menos, sabermos uma fórmula para fazê-la.

O método resume-se na aplicação de divisões sucessivas por 2, mantendo o resto como o dígito binário e o resultado como o próximo número a ser dividido.

Como exemplo podemos usar o número decimal 43.

43/2=21 e o resto é 1;

21/2=10 e o resto é 1;

10/2=5 e o resto é 0;

5/2=2 e o resto é 1;

2/2=1 e o resto é 0;

1/2=0 e o resto é 1.

Para construir o equivalente binário de 43, vamos utilizar os restos obtidos de baixo para cima, assim temos 101011.

Sistema hexadecimal

Na base hexadecimal temos 16 dígitos, que vão de 0 a 9 e da letra A até a F, estas letras representam os números de 10 a 15. Portanto contamos: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E, e F.

A conversão entre números binários e hexadecimais é fácil. A primeira coisa a fazer é dividir o número binário em grupos de 4 bits, começando da direita para a esquerda. Se no grupo mais à direita sobrarem dígitos, completamos com zeros.

Tomando como exemplo o número binário 101011, vamos dividi-lo em grupos de 4 bits:

10;1011

A seguir, tomamos cada grupo como um número independente e consideramos o seu valor decimal:

0010=2;

1011=11;

Entretanto, observa-se que não podemos representar este número como 211, isto seria um erro, uma vez que os números em hexadecimal maiores que 9 e menores que 16 são representados pelas letras A,B,...,F. Logo, obtemos como resultado:

2Bh, onde o "h" representa a base hexadecimal.

Para a conversão de um número hexadecimal em binário é apenas necessário inverter os passos: usamos o primeiro dígito hexadecimal e convertemo-lo para binário, a seguir o segundo, e assim por diante.

Métodos de representação de dados num computador

Código ASCII

ASCII significa American Standard Code for Information Interchange. Este código contém as letras do alfabeto, dígitos decimais de 0 a 9 e alguns símbolos adicionais como um número binário de 7 bits, tendo o oitavo bit em 0, ou seja, desligado.

Deste modo, cada letra, dígito ou caracter especial ocupa 1 byte na memória do computador.

Podemos observar que este método de representação de dados é muito ineficiente no aspeto numérico, uma vez que no formato binário 1 byte não é suficiente para representar números de 0 a 255, com o ASCII podemos representar apenas um dígito.

Devido a esta ineficiência, o código ASCII é usado, principalmente, para a representação de textos.

Método BCD

BCD significa Binary Coded Decimal.

Neste método grupos de 4 bits são usados para representar cada dígito decimal de 0 a 9. Com este método podemos representar 2 dígitos por byte de informação.

Vemos que este método vem a ser muito mais prático para representação numérica do que o código ASCII. Embora ainda menos prático do que o binário, com o método BCD podemos representar dígitos de 0 a 99. Com o binário, vemos que o alcance é maior, de 0 a 255.

Este formato (BCD) é principalmente usado na representação de números grandes, aplicações comerciais, devido às suas facilidades de operação.

Representação de ponto flutuante

Esta representação é baseada em notação científica, isto é, representar um número em 2 partes: a sua base e o seu expoente.

Por exemplo o número decimal 1234000, é representado como 1.234*10^6, observamos que o expoente irá indicar o número de casas que o ponto decimal deve ser deslocado para a direita, a fim de obtermos o número original.

O expoente negativo, por outro lado, indica o número de casas que o ponto decimal deve se deslocar para a esquerda.

Uso do Programa DEBUG

Processo de criação de programas

Para a criação de programas são necessários os seguintes passos:

- Desenvolvimento do algoritmo, etapa em que o problema a ser solucionado é estabelecido e a melhor solução é proposta, criação de diagramas esquemáticos relativos à melhor solução proposta.
- Codificação do algoritmo, o que consiste em escrever o programa em alguma linguagem de programação; linguagem assembly neste caso específico, tomando como base a solução proposta no passo anterior.

- A transformação para a linguagem de máquina, ou seja, a criação do programa objeto, escrito como uma sequência de zeros e uns que podem ser interpretados pelo processador.
- O último estágio é a eliminação de erros detetados no programa na fase de teste. A correção normalmente requer a repetição de todos os passos, com observação atenta.

Registadores da CPU

Para o propósito didático, vamos focar registadores de 16 bits. A CPU possui 4 registadores internos, cada um de 16 bits. São eles AX, BX, CX e DX. São registadores de uso geral e também podem ser usados como registadores de 8 bits. Para tal devemos referenciá-los como, por exemplo, AH e AL, que são, respetivamente, o byte high e o low do registador AX. Esta nomenclatura também se aplica para os registadores BX, CX e DX.

Os registadores, segundo seus respetivos nomes:

- AX Registador Acumulador
- BX Registador Base
- CX Registador Contador
- DX Registador de Dados
- DS Registador de Segmento de Dados
- ES Registador de Segmento Extra
- SS Registador de Segmento de Pilha
- CS Registador de Segmento de Código
- BP Registador Apontador da Base
- SI Registador de Índice Fonte
- DI Registador de Índice Destino
- SP Registador Apontador de Pilha
- IP Registador Apontador da Próxima Instrução
- F Registador de Flag

Programa Debug

Para a criação de um programa em assembler existem 2 opções: usar o TASM - Turbo Assembler da Borland, ou o DEBUGGER. Nesta primeira seção vamos usar o debug, uma vez que podemos encontrá-lo em qualquer PC com o MS-DOS.

Debug pode apenas criar ficheiros com a extensão ".COM", e por causa das características deste tipo de programa, eles não podem exceder os 64 Kb, e também devem iniciar no endereço de memória 0100H dentro do segmento específico.

É importante observar isso, pois deste modo os programas ".COM" não são realocáveis.

Os principais comandos do programa debug são:

- A Montar instruções simbólicas em código de máquina
- D Mostrar o conteúdo de uma área da memória
- E Entrar dados na memória, iniciando num endereço específico
- G Executar um programa executável na memória
- N Dar nome a um programa
- P Proceder, ou executar um conjunto de instruções relacionadas
- Q Sair do programa debug
- R Mostrar o conteúdo de um ou mais registadores
- T Executar passo a passo as instruções
- U Desmontar o código de máquina em instruções simbólicas
- W Gravar um programa em disco

É possível visualizar os valores dos registadores internos da CPU usando o programa Debug. Debug é um programa que faz parte do pacote do DOS, e pode ser encontrado normalmente no diretório C:\DOS. Para iniciá-lo, basta digitar Debug na linha de comando:

```
C:/>Debug [Enter]
-
```

Irmos notar então a presença de um hífen no canto inferior esquerdo do ecrã. Não se espante, este é o prompt do programa. Para visualizar o conteúdo dos registadores, experimente:

```
-r[Enter]
```

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE B P = 0 0 0 0

SI=0000 DI=0000 DS=0D62 ES=0D62 SS=0D62 C S = 0 D 6 2

IP=0100 NV UP EI PL NZ NA PO NC 0D62:0100 2E CS: 0D62:0101 803ED3DF00 C M P

BYTE PTR [DFD3],00 CS:DFD3=03

É mostrado o conteúdo de todos os registadores internos da CPU; um modo alternativo para visualizar um único registador é usar o comando "r" seguido do parâmetro que faz referência ao nome do registador:

-rbx

BX 0000

:

Esta instrução mostrará o conteúdo do registador BX e mudará o indicador do Debug de "-" para ":"

Quando o prompt assim se tornar, significa que é possível, embora não obrigatória, a mudança do valor contido no registador, bastando digitar o novo valor e pressionar [Enter]. Se simplesmente pressionar-mos [Enter] o valor antigo mantém-se.

Estrutura Assembly

Nas linhas do código em Linguagem Assembly há duas partes: a primeira é o nome da instrução a ser executada; a segunda, os parâmetros do comando. Por exemplo:

add ah bh

Aqui "add" é o comando a ser executado, neste caso uma adição, e "ah" bem como "bh" são os parâmetros.

Por exemplo:

mov al, 25

No exemplo acima, estamos a utilizar a instrução mov, que significa mover o valor 25 para o registador al.

O nome das instruções nesta linguagem é constituído de 2, 3 ou 4 letras. Estas instruções são chamadas de mnemónicos ou códigos de operação, representando a função que o processador executará.

Às vezes instruções aparecem assim:

add al,[170]

Os parenteses retos no segundo parâmetro indica-nos que vamos trabalhar com o conteúdo da célula de memória de número 170, ou seja, com o valor contido no endereço 170 da memória e não com o valor 170, isto é conhecido como "endereçamento direto".

Criação de um programa simples em assembly

Vamos, então, criar um programa para ilustrar o que vimos até agora. Adicionaremos dois valores:

O primeiro passo é iniciar o Debug, o que já vimos como fazer anteriormente.

Para montar um programa no Debug, é usado o comando "a" (assembly); quando usamos este comando, podemos especificar um endereço inicial para o nosso programa como o parâmetro, mas é opcional. No caso de omissão, o endereço inicial é o especificado pelos registadores CS:IP, geralmente 0100h, o local em que programas com extensão ".COM" devem iniciar. E será este o local que usaremos, uma vez que o Debug só pode criar este tipo de programa.

Embora neste momento não seja necessário darmos um parâmetro ao comando "a", isso é recomendável para evitar problemas, logo:

```
a 100           [enter]
mov ax,0002     [enter]
mov bx,0004     [enter]
add ax,bx       [enter]
nop[enter]      [enter]
```

O que o programa faz? Move o valor 0002 para o registador ax, move o valor 0004 para o registador bx, adiciona o conteúdo dos registadores ax e bx, guardando o resultado em ax e finalmente a instrução nop (nenhuma operação) finaliza o programa.

No programa debug, o ecrã a aparecer será parecido com:

```
C:\>debug
-a 100
0D62:0100 mov ax,0002
0D62:0103 mov bx,0004
```

0D62:0106 add ax,bx

0D62:0108 nop

0D62:0109

Entramos com o comando “t” para executar passo a passo as instruções:

AX=0002 BX=0000 CX=0000 DX=0000 SP=FFEE B P = 0 0 0 0

SI=0000 DI=0000 DS=0D62 ES=0D62 SS=0D62 CS=0D62

IP=0103 NV UP EI PL NZ NA PO NC 0D62:0103 BB0400 MOV BX,0004

Vemos o valor 0002 no registador AX. Teclamos “t” para executar a segunda instrução:

-t

AX=0002 BX=0004 CX=0000 DX=0000 SP=FFEE B P = 0 0 0 0

SI=0000 DI=0000 DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0106

NV UP EI PL NZ NA PO NC 0D62:0106 01D8 ADD AX,BX

Teclando “t” novamente para ver o resultado da instrução add:

-t

AX=0006 BX=0004 CX=0000 DX=0000 SP=FFEE B P = 0 0 0 0

SI=0000 DI=0000 DS=0D62 ES=0D62 SS=0D62 CS=0D62

IP=0108 NV UP EI PL NZ NA PE NC 0D62:0108 90 NOP

A possibilidade dos registadores conterem valores diferentes existe, mas AX e BX devem conter os mesmos valores acima descritos.

Para sair do Debug usamos o comando “q” (quit).

Guardar e executar os programas

Não seria prático ter que digitar o programa de cada vez que iniciássemos o Debug. Ao invés disso, podemos armazená-lo no disco. Só que o mais interessante nisto é que um simples comando de guardar cria um ficheiro com a extensão “.COM”, ou seja, executável - sem precisarmos efetuar os processos de montagem e ligação, como veremos posteriormente com o TASM.

Eis os passos para guardar um programa que já esteja na memória:

- Obter o tamanho do programa subtraindo o endereço final do endereço inicial, naturalmente será no sistema hexadecimal.
- Dar um nome ao programa.

- Colocar o tamanho do programa no registador CX.
- Mandar o debug gravar o programa em disco.

Usando como exemplo o seguinte programa, vamos aclarar a ideia de como realizar os passos acima descritos:

```
0C1B:0100 mov ax,0002
0C1B:0103 mov bx,0004
0C1B:0106 add ax,bx
0C1B:0108 int 20
0C1B:010A
```

Para obter o tamanho de um programa, o comando "h" é usado, já que ele nos mostra a adição e subtração de dois números em hexadecimal. Para obter o tamanho do programa em questão, damos como parâmetro o valor do endereço final do nosso programa (10A), e o endereço inicial (100). O primeiro resultado mostra-nos a soma dos endereços, o segundo, a subtração.

```
-h 10a 100
020a 000a
```

O comando "n" permite-nos nomear o programa.

```
-n test.com
```

O comando "rcx" permite-nos mudar o conteúdo do registador CX para o valor obtido como tamanho do ficheiro com o comando "h", neste caso 000a.

```
-rcx
CX 0000
:000a
```

Finalmente, o comando "w" grava o nosso programa no disco, indicando quantos bytes gravou.

```
-w
Writing 000A bytes
```

Já sabemos guardar um ficheiro, e para executa-lo, são necessários dois passos :

- Dar o nome do ficheiro a ser executado.
- Executa-lo utilizando o comando "l" (load).

Para obter o resultado correto destes passos, é necessário que o programa acima já esteja criado.

Dentro do Debug, escrevemos o seguinte:

```
-n test.com
-l
-u 100 109
0C3D:0100 B80200 MOV AX,0002
0C3D:0103 BB0400 MOV BX,0004
0C3D:0106 01D8 ADD AX,BX
0C3D:0108 CD20 INT 20
```

O último comando “u” é usado para verificar que o programa foi carregado na memória. O que ele faz é desmontar o código e mostrá-lo em assembly. Os parâmetros indicam ao Debug os endereços inicial e final a serem desmontados.

O Debug executa sempre os programas na memória no endereço 100h, conforme já comentamos.

Programação Assembly

Construindo programas em Assembly

Software necessário

Para que possamos criar um programa, precisamos de algumas ferramentas:

1. Um editor para criar o programa fonte.
2. Um montador, um programa que irá transformar o nosso programa fonte num programa objeto.
3. Um linker (ligador) que irá gerar o programa executável a partir do programa objeto.

O editor pode ser qualquer um que dispusermos. O montador será o TASM macro assembler da Borland, e o linker será o TLINK, também da Borland.

Devemos criar os programas fonte com a extensão .ASM para que o TASM reconheça e o transforme no programa objeto, um "formato intermediário" do programa, assim chamado porque ainda não é um programa executável e tão pouco um programa fonte.

O linker gera a partir de um programa ".OBJ", ou da combinação de vários deles, um programa executável, cuja extensão é normalmente ".EXE", embora possa ser ".COM" dependendo da forma como for montado e ligado.

Programação Assembly

Para construirmos os programas com o TASM, devemos estruturar o programa fonte de forma diferenciada do que fazíamos com o programa debug.

É importante incluir as seguintes diretivas assembly:

.MODEL SMALL - Define o meio de memória a usar em nosso programa

.CODE - Define as instruções do programa, relacionado ao segmento de código

.STACK - Reserva espaço de memória para as instruções de programa na pilha

END - Finaliza um programa assembly

Começando a programar

1. *Primeiro passo*

Usar qualquer editor para criar o programa fonte. Começamos com as seguintes linhas:

Exemplo 1

```
; use ; para fazer comentários em programas assembly
        .MODEL SMALL        ;modelo de memória
        .STACK              ;espaço de memória para instruções do programa na pilha
        .CODE               ;as linhas seguintes são instruções do programa
        mov ah,01h          ;move o valor 01h para o registador ah
        mov cx,07h          ;move o valor 07h para o registador cx
        int 10h             ;interrupção 10h
        mov ah,4ch          ;move o valor 4ch para o registador ah
        int 21h             ;interrupção 21h
        END                 ;finaliza o código do programa
```

Este programa assembly muda o tamanho do cursor.

2. *Segundo passo*

Guardar o ficheiro com o seguinte nome: “exam1.asm”

Não esquecer de guarda-lo no formato ASCII.

3. *Terceiro passo*

Usar o programa TASM para construir o programa objeto.

Exemplo:

```
C:\>tasm exam1.asm
Turbo Assembler Version 2.0 Copyright (c) 1988, 1990 Borland International
Assembling file: exam1.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 471k
```

O TASM só pode criar programas no formato “.OBJ”, que ainda não pode ser executado...

4. *Quarto passo*

Usar o programa TLINK para criar o programa executável.

Exemplo:

```
C:\>tlink exam1.obj
Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International
C:\>
```

Onde exam1.obj é o nome do programa intermediário, ".OBJ". O comando acima gera diretamente o ficheiro com o nome do programa intermediário e a extensão ".EXE". É opcional a colocação da extensão ".obj" no comando.

5. *Quinto passo*

Executar o programa executável criado.

```
C:\>exam1[enter]
```

De lembrar, que este programa assembly muda o tamanho do cursor no DOS.

Processo Assembly

Segmentos

A arquitetura dos processadores x86 força-nos a usar segmentos de memória para gerir a informação, o tamanho destes segmentos é de 64Kb.

A razão de ser destes segmentos é que, considerando que o tamanho máximo de um número que o processador pode gerir é dado por uma palavra de 16 bits ou registador, assim não seria possível aceder a mais do que 65536 locais da memória utilizando apenas um destes registadores. Mas agora, se a memória do PC é dividida em grupos de segmentos, cada um com 65536 locais, e podemos usar um endereço ou registador exclusivo para encontrar cada segmento, e ainda fazemos cada endereço de um slot específico com dois registadores, é nos possível aceder à quantidade de 4294967296 bytes de memória, que é, atualmente, a maior memória que podemos instalar num PC.

Desta forma, para que o montador seja capaz de gerir os dados, faz-se necessário que cada informação ou instrução se encontre na área correspondente ao seu segmento. O endereço do segmento é fornecido ao montador pelos registadores DS, ES, SS e CS.

Lembrando um programa no Debug, observe:

1CB0:0102 MOV AX,BX

O primeiro número 1CB0, corresponde ao segmento de memória que está a ser usado, o segundo é uma referência ao endereço dentro do segmento, é um deslocamento dentro do segmento offset.

O modo utilizado para indicar ao montador com quais segmentos vamos trabalhar é fazendo uso das diretivas:

.CODE

.DATA

.STACK.

O montador ajusta o tamanho dos segmentos tomando como base o número de bytes que cada instrução assembly precisa, já que seria um desperdício de memória usar segmentos inteiros. Por exemplo, se um programa precisa de apenas 10Kb para armazenar dados, o segmento de dados seria apenas de 10Kb e não de 64Kb, como poderia acontecer se feito manualmente.

Tabelas de Equivalência

Cada uma das partes numa linha de código assembly é conhecida como token, por exemplo:

MOV AX,Var

Aqui temos três tokens, a instrução MOV, o operador AX e o operador VAR. O que o montador faz para gerir o código OBJ é ler cada um dos tokens e procurar a equivalência em código de máquina em tabelas correspondentes, seja de palavras reservadas, tabela de códigos de operação, tabela de símbolos, tabela de literais, onde o significado dos mnemônicos e os endereços dos símbolos que usamos serão encontrados.

A maioria dos montadores são de duas passagens. Em síntese na primeira passagem temos a definição dos símbolos, ou seja, são associados endereços a todas as instruções do programa. Seguindo este processo, o assembler lê MOV e procura-o na tabela de códigos de operação para encontrar o seu equivalente na linguagem de máquina. Da mesma forma ele lê AX e encontra-o na tabela correspondente como sendo um registador. O processo para Var é um pouco diferenciado, o montador verifica que ela

não é uma palavra reservada, então procura na tabela de símbolos, lá encontrando-a ele designa o endereço correspondente, mas se não encontrar ele insere-a na tabela para que ela possa receber um endereço na segunda passagem. Ainda na primeira passagem é executado parte do processamento das diretivas, é importante notar que as diretivas não criam código objeto. Na passagem dois são montadas as instruções, traduzindo os códigos de operação e procurando os endereços, e é gerado o código objeto.

Há símbolos que o montador não consegue encontrar, uma vez que podem ser declarações externas. Neste caso o linker entra em ação para criar a estrutura necessária a fim de ligar as diversas possíveis partes de código, dizendo ao loader que o segmento e o token em questão são definidos quando o programa é carregado e antes de ser executado.

Pequenos programas Assembly

Exemplo 2

1. *Primeiro passo*

Use qualquer editor e crie o seguinte:

```
;exemplo2
.model small
.stack
.code
mov ah,2h       ;move o valor 2h para o registador ah
mov dl,2ah      ;move o valor 2ah para o registador dl
                ;(é o valor ASCII do caractere *)
int 21h         ;interrupção 21h
mov ah,4ch      ;função 4ch, sai para o sistema operacional
int 21h         ;interrupção 21h
end             ;finaliza o programa
```

2. *Segundo passo*

Guardar o ficheiro com o nome: exam2.asm

Não esquecer de guardar em formato ASCII.

3. *Terceiro passo*

Usar o programa TASM para construir o programa objeto.

```
C:\>tasm exam2.asm
Turbo Assembler Version 2.0 Copyright (c) 1988, 1990 Borland International
Assembling file: exam2.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 471k
```

4. *Quarto passo*

Usar o programa TLINK para criar o programa executável.

```
C:\>tlink exam2.obj
Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International
C:\>
```

5. *Quinto passo*

Executar o programa:

```
C:\>exam2[enter]
*
C:\>
```

Este programa imprime o caracter * no ecrã.

Tipos de instruções

Movimento de dados

Em qualquer programa há necessidade de se moverem dados na memória e em registadores da CPU; há vários modos de se fazer: podem-se copiar os dados da memória para algum registador, de registador para registador, de um registador para a pilha, da pilha para um registador, transmitir dados para um dispositivo externo e vice-versa.

Este movimento de dados está sujeito a regras e restrições, entre elas:

- Não é possível mover dados de um local da memória para outro diretamente; é necessário primeiro mover o dado do local de origem para um registador e então do registador para o local de destino.
- Não é possível mover uma constante diretamente para um registador de segmento; primeiro deve-se mover para um registador.

É possível mover blocos de dados através de instruções movs, que copia uma cadeia de bytes ou palavras; movsb copia n bytes de um local para outro; e movsw copia n palavras. A última das duas instruções toma os valores dos endereços definidos por DS:SI como o grupo de dados a mover e ES:DI como a nova localização dos dados.

Para mover dados há também estruturas chamadas pilhas, onde o dado é introduzido com a instrução push e é extraído com a instrução pop.

Numa pilha o primeiro dado a entrar é o último a sair, por exemplo:

```
PUSH AX
PUSH BX
PUSH CX
```

Para retornar os valores da pilha referentes a cada registador é necessário seguir a seguinte ordem:

```
POP CX
POP BX
POP AX
```

Para a comunicação com dispositivos externos o comando de saída é usado para o envio de informações a uma porta e o comando de entrada é usado para receber informação de uma porta.

A sintaxe do comando de saída:

```
OUT DX,AX
```

Onde DX contém o valor da porta que será usada para a comunicação e AX contém a informação que será enviada.

A sintaxe do comando de entrada:

```
IN AX,DX
```

Onde AX é o registador onde a informação será armazenada e DX contém o endereço da porta de onde chegará a informação.

Operações Lógicas e Aritméticas

As instruções de operações lógicas são: and, not, or e xor. Elas trabalham a nível de bits nos seus operadores.

Para verificar o resultado das operações usamos as instruções cmp e test.

As instruções usadas para operações algébricas são:

- Para adição add
- Para subtração sub
- Para multiplicação mul
- Para divisão div

Quase todas as instruções de comparação são baseadas na informação contida no registador de flag. Normalmente os flags do registador que podem ser manuseados diretamente pelo programador e são os da direção de dados DF, usados para definir as operações sobre cadeias. Uma outra que pode também ser manuseado é o flag IF através das instruções sti e cli, para ativar e desativar as interrupções.

Saltos, Loops e Procedimentos

Saltos incondicionais na escrita de programas em linguagem assembly são dados pela instrução;

jmp;

Um salto é usado para modificar a sequência da execução das instruções de um programa, enviando o controlo ao endereço indicado, ou seja, o registador contador de programa recebe este novo endereço.

Um loop, também conhecido como interação, é a repetição de um processo um certo número de vezes até atingir a condição de paragem.

Questionário

1. Os registadores são grupos de 8 ou 16 flip-flops. Qual a função dos flip-flops?

2. Um grupo de 16 bits é conhecido como palavra. Como pode ser dividida essa palavra?

3. Converta os seguintes números para as bases binária, octal e hexadecimal:
   a. 10_{10}
   b. 64_{10}
   c. 121_{10}
   d. 1255_{10}
   e. 512_{10}
   f. 497_{10}

4. Quais são os métodos de representação de dados num computador? Explique pelo menos um dele.

5. Existem diversos registadores da CPU. Dos seguintes apresentados, fça a respetiva legrnda de cada um.

   AX _____
   BX _____
   DX _____
   SS _____
   CS _____
   SI _____
   DI _____
   SP _____
   F _____

6. No Debug do Windows quais são os comandos que permitem Dar nome a um programa e mostrar o conteúdo de um ou mais registadores?

7. O que faz a seguinte instrução?

   mov al, 30

8. Descreva o funcionamento do programa que se segue.

   a 100 [enter]
   mov ax,0003 [enter]
   mov bx,0005 [enter]
   add ax,bx [enter]
   nop[enter] [enter]

9. Faça corresponder as seguintes diretivas com o seu significado.

.STACK	Define o meio de memória a usar em nosso programa
END	Define as instruções do programa, relacionado ao segmento de código
.MODEL SMALL	Reserva espaço de memória para as instruções de programa na pilha
.CODE	Finaliza um programa assembly

10. Quais as instruções mara mover dados numa pilha de dados?

11. Diga que instruções são utilizadas para operações algébricas.

12. As instruções jmp e loop são para ?

    a. Jmp - Apagar dados ; loop – repetição de um processo
    b. Jmp - Saltos incondicionais ; loop – repetição de um processo
    c. loop - Saltos incondicionais ; jmp – repetição de um processo
    d. loop - Apagar dados ; jmp – repetição de um processo

Instruções Assembly

Instruções de operação de dados

Instruções de transferência

São usadas para mover o conteúdo dos operadores. Cada instrução pode ser usada com diferentes modos de endereçamento.

MOV

MOVS (MOVSB) (MOVSW)

INSTRUÇÃO MOV

Propósito: Transferência de dados entre células de memória, registadores e o acumulador.

Sintaxe:

MOV Destino, Fonte

Destino é o lugar para onde o dado será movido e Fonte é o lugar onde o dado está.

Os diferentes movimentos de dados permitidos para esta instrução são:

*Destino: memória.	Fonte: acumulador
*Destino: acumulador.	Fonte: memória
*Destino: registador de segmento.	Fonte: memória/registador
*Destino: memória/registador.	Fonte: registador de segmento
*Destino: registador.	Fonte: registador
*Destino: registador.	Fonte: memória
*Destino: memória.	Fonte: registador
*Destino: registador.	Fonte: dado imediato
*Destino: memória.	Fonte: dado imediato

Exemplo:

```
MOV AX,0006h
MOV BX,AX
MOV AX,4C00h
INT 21h
```

Este pequeno programa move o valor 0006h para o registador AX, depois move o conteúdo de AX (0006h) para o registador BX, e finalmente move o valor 4C00h para o registador AX para terminar a execução com a opção 4C da interrupção 21h.

INSTRUÇÕES MOVS (MOVSB) (MOVSW)

Propósito: Mover byte ou cadeias de palavra da fonte, endereçada por SI, para o destino endereçado por DI.

Sintaxe:

MOVS

Este comando não precisa de parâmetros uma vez que toma como endereço fonte o conteúdo do registador SI e como destino o conteúdo de DI. A seguinte sequência de instruções ilustra isso:

```
MOV SI, OFFSET VAR1
MOV DI, OFFSET VAR2
MOVS
```

Primeiro inicializamos os valores de SI e DI com os endereços das variáveis VAR1 e VAR2 respetivamente, então após a execução de MOVS o conteúdo de VAR1 é copiado para VAR2.

As instruções MOVSB e MOVSW são usadas do mesmo modo que MOVS, a primeira move um byte e a segunda move uma palavra.

Instruções de carga

São instruções específicas para registadores, usadas para carregar bytes ou cadeias de bytes num registador.

LODS (LODSB) (LODSW)

LAHF

LDS

LEA

LES

INSTRUÇÕES LODS (LODSB) (LODSW)

Propósito: Carregar cadeias de um byte ou uma palavra para o acumulador.

Sintaxe:

LODS

Esta instrução toma a cadeia encontrada no endereço especificado por SI, e carrega-a para o registador AL (ou AX) e adiciona ou subtrai, dependendo do estado de DF, para SI se é uma transferência de bytes ou de palavras.

```
MOV SI, OFFSET VAR1
LODS
```

Na primeira linha vemos a carga do endereço de VAR1 em SI e na segunda é ocupado o conteúdo daquele local para o registador AL.

Os comandos LODSB e LODSW são usados do mesmo modo, o primeiro carrega um byte e o segundo uma palavra (usa todo o registador AX).

INSTRUÇÃO LAHF

Propósito: Transferir o conteúdo dos flags para o registador AH.

Sintaxe:

LAHF

Esta instrução é útil para verificar o estado dos flags durante a execução do nosso programa.

Os flags são deixados na seguinte ordem dentro do registador:

SF ZF ?? AF ?? PF ?? CF

O "??" significa que haverá um valor indefinido naqueles bits.

INSTRUÇÃO LDS

Propósito: Carregar o registador de segmento de dados.

Sintaxe:

LDS destino, fonte

O operador fonte deve ser uma double word na memória. A palavra associada com o maior endereço é transferida para DS, por outras palavras isto é ocupado como o endereço de segmento. A palavra associada com o menor endereço é o endereço de deslocamento e é depositada no registador indicado como destino.

INSTRUÇÃO LEA

Propósito: Carregar o endereço do operador fonte.

Sintaxe:

LEA destino,fonte

O operador fonte deve estar localizado na memória, e o seu deslocamento é colocado no registador de índice ou ponteiro especificado no destino.

Para ilustrar uma das facilidades que temos com este comando, vejamos:

MOV SI,OFFSET VAR1

É equivalente a:

LEA SI,VAR1

É muito provável que para o programador é muito mais fácil criar programas grandes usando este último formato.

INSTRUÇÃO LES

Propósito: Carregar o registador de segmento extra

Sintaxe:

LES destino,fonte

O operador fonte deve ser uma palavra dupla na memória. O conteúdo da palavra com endereço maior é interpretado como o endereço do segmento e é colocado em ES. A palavra com endereço menor é o endereço do deslocamento e é colocada no registador especificado no parâmetro de destino.

Instruções de manipulação da pilha

Estas instruções permitem usar a pilha para armazenar ou recuperar dados.

POP

POPF

PUSH

PUSHF

INSTRUÇÃO POP

Propósito: Recuperar uma parte de informação da pilha.

Sintaxe:

POP destino

Esta instrução transfere o último valor armazenado na pilha para o operador de destino, e incrementa de dois em dois o registador SP.

Este incremento é duplo porque a pilha vai do mais alto endereço de memória para o mais baixo, e a pilha trabalha apenas com palavras, 2 bytes, logo deve ser de dois em dois o incremento de SP, na realidade o dois está a ser subtraído do tamanho real da pilha.

INSTRUÇÃO POPF

Propósito: Extrair os flags armazenados na pilha.

Sintaxe:

POPF

Este comando transfere os bits da palavra armazenada na parte mais alta da pilha para registador de flag.

O modo da transferência é como se segue:

BIT	FLAG
0	CF
2	PF
4	AF
6	ZF
7	SF
8	TF
9	IF
10	DF
11	OF

Os locais dos bits são os mesmos para o uso da instrução PUSHF.

Uma vez feita a transferência o registador SP é feito um incremento de dois, conforme vimos anteriormente.

INSTRUÇÃO PUSH

Propósito: Coloca uma palavra na pilha.

Sintaxe:

PUSH fonte

A instrução PUSH decrementa de dois em dois o valor de SP e então transfere o conteúdo do operador fonte para o novo endereço resultante no registador recém modificado.

O decremento no endereço é duplo pelo fato de que quando os valores são adicionados à pilha, que cresce do maior para o menor endereço, logo quando subtraímos de dois o registador SP o que fazemos é incrementar o tamanho da pilha em dois bytes, que é a única quantidade de informação que a pilha pode manusear em cada entrada e saída.

INSTRUÇÃO PUSHF

Propósito: Colocar os valores dos flags na pilha.

Sintaxe:

PUSHF

Este comando decrementa de dois em dois o valor do registador SP e então o conteúdo do registador de flag é transferido para a pilha, no endereço indicado por SP.

Os flags são armazenados na memória da mesma forma que o comando POPF.

Instruções lógicas e aritméticas

Instruções lógicas

São usadas para realizar operações lógicas nos operadores.

AND

NEG

NOT

OR

TEST

XOR

INSTRUÇÃO AND

Propósito: Realiza a conjunção de operadores bit a bit.

Sintaxe:

AND destino,fonte

Com esta instrução a operação lógica "y" para ambos os operadores é usada como na tabela:

Fonte	Destino	Destino
1	1	1
1	0	0
0	1	0
0	0	0

O resultado desta operação é armazenado no operador de destino.

INSTRUÇÃO NEG

Propósito: Gera o complemento de 2.

Sintaxe:

NEG destino

Esta instrução gera o complemento de 2 do operador destino e armazena-o no mesmo operador. Por exemplo, se AX armazena o valor 1234H, então:

NEG AX

Isto fará com o que o valor EDCCH fique armazenado no registador AX.

INSTRUÇÃO NOT

Propósito: Faz a negação do operador de destino bit a bit.

Sintaxe:

NOT destino

O resultado é armazenado no mesmo operador de destino.

INSTRUÇÃO OR

Propósito: Realiza um OU lógico.

Sintaxe:

OR destino,fonte

A instrução OR, faz uma disjunção lógica bit a bit dos dois operadores:

Fonte	Destino	Destino
1	1	1
1	0	1
0	1	1
0	0	0

INSTRUÇÃO TEST

Propósito: Compara logicamente os operadores.

Sintaxe:

TEST destino,fonte

Realiza uma conjunção, bit a bit, dos operadores, mas difere da instrução AND, uma vez que não coloca o resultado no operador de destino. Tem efeito sobre o registador de flag.

INSTRUÇÃO XOR

Propósito: Realiza um OU exclusivo.

Sintaxe:

XOR destino,fonte

Esta instrução realiza uma disjunção exclusiva de dois operadores bit a bit.

Fonte	Destino	Destino
1	1	0
0	0	1
0	1	1
0	0	0

Instruções aritméticas

São usadas para realizar operações aritméticas nos operadores.

ADC

ADD

DIV

IDIV

MUL

IMUL

SBB

SUB

INSTRUÇÃO ADC

Propósito: Efetuar a soma entre dois operandos com carry.

Sintaxe:

ADC destino,fonte

Esta instrução efetua a soma entre dois operandos, mais o valor do flag CF, existente antes da operação. Apenas o operando destino e os flags são afetados.

O resultado é armazenado no operador de destino.

INSTRUÇÃO ADD

Propósito: Adição de dois operadores.

Sintaxe:

ADD destino,fonte

Esta instrução adiciona dois operadores e armazena o resultado no operador destino.

INSTRUÇÃO DIV

Propósito: Divisão sem sinal.

Sintaxe:

DIV fonte

O divisor pode ser um byte ou uma palavra e é o operador que é dado na instrução.

Se o divisor é de 8 bits, o registador AX de 16 bits é ocupado como dividendo e se o divisor é de 16 bits, o par de registadores DX:AX será ocupado como dividendo, tomando a palavra alta de DX e a baixa de AX.

Se o divisor for um byte, então o quociente será armazenado no registador AL e o resto em AH. Se for uma palavra, então o quociente é armazenado em AX e o resto em DX.

INSTRUÇÃO IDIV

Propósito: Divisão com sinal.

Sintaxe:

IDIV fonte

Consiste basicamente na instrução DIV, diferencia-se apenas por realizar a operação com sinal.

Para os resultados são usados os mesmos registadores da instrução DIV.

INSTRUÇÃO MUL

Propósito: Multiplicação com sinal.

Sintaxe:

MUL fonte

Esta instrução realiza uma multiplicação não sinalizada entre o conteúdo do acumulador AL ou AX pelo operando-fonte, devolvendo o resultado no acumulador AX caso a operação tenha envolvido AL com um operando de 8 bits, ou em DX e AX caso a operação tenha envolvido AX e um operando de 16 bits.

INSTRUÇÃO IMUL

Propósito: Multiplicação de dois números inteiros com sinal.

Sintaxe:

IMUL fonte

Esta instrução faz o mesmo que a anterior, difere apenas pela inclusão do sinal.

Os resultados são mantidos nos mesmos registadores usados pela instrução MUL.

INSTRUÇÃO SBB

Propósito: Subtração com carry.

Sintaxe:

SBB destino,fonte

Esta instrução subtrai os operadores e subtrai um do resultado se CF está ativo. O operador fonte é sempre subtraído do destino.

Este tipo de subtração é usado quando se trabalha com quantidades de 32 bits.

INSTRUÇÃO SUB

Propósito: Subtração.

Sintaxe:

SUB destino,fonte

Esta instrução subtrai o operador fonte do destino.

Instruções de controle de processos

Instruções de salto

Usadas para transferir o processo de execução do programa para o operador indicado.

JMP

JA (JNBE)

JAE (JNBE)

JB (JNAE)

JBE (JNA)

JE (JZ)

JNE (JNZ)

JG (JNLE)

JGE (JNL)

JL (JNGE)

JLE (JNG)

JC

JNC

JNO

JNP (JPO)

JNS

JO

JP (JPE)

JS

INSTRUÇÃO JMP

Propósito: Salto incondicional.

Sintaxe:

JMP destino

Esta instrução é usada para desviar o curso do programa sem ter em conta as condições atuais dos flags ou dos dados.

INSTRUÇÃO JA (JNBE)

Propósito: Salto condicional.

Sintaxe:

JA símbolo

Após uma comparação este comando salta se não é igual.

Isto quer dizer que o salto só é feito se o flag CF ou o flag ZF estão desativados, ou seja, se um dos dois for zero.

INSTRUÇÃO JAE (JNB)

Propósito: Salto condicional.

Sintaxe:

JAE símbolo

A instrução salta se está em um, se for igual ou se for zero negado.

O salto é feito se CF estiver desativado.

INSTRUÇÃO JB (JNAE)

Propósito: Salto condicional.

Sintaxe:

JB símbolo

A instrução salta se for zero, se for um negado ou se for igual.

O salto é feito se CF estiver ativo.

INSTRUÇÃO JBE (JNA)

Propósito: Salto condicional.

Sintaxe:

JBE símbolo

A instrução salta se for zero, ou igual ou se for em um negado.

O salto é feito se CF ou ZF estão ativos, ou seja, se um deles for 1.

INSTRUÇÃO JE (JZ)

Propósito: Salto condicional.

Sintaxe:

JE símbolo

A instrução salta se for igual ou se for zero.

O salto é feito se ZF está ativo.

INSTRUÇÃO JNE (JNZ)

Propósito: Salto condicional.

Sintaxe:

JNE símbolo

A instrução salta se for zero.

O salto é feito se ZF estiver desativado.

INSTRUÇÃO JG (JNLE)

Propósito: Salto condicional, e o sinal é ocupado.

Sintaxe:

JG símbolo

A instrução salta se for maior, se for não maior ou se for igual.

O salto ocorre se ZF = 0 ou se OF = SF.

INSTRUÇÃO JGE (JNL)

Propósito: Salto condicional, e o sinal é ocupado.

Sintaxe:

GE símbolo

A instrução salta se for maior, se for menor que ou se for igual.

O salto é feito se SF = OF.

INSTRUÇÃO JL (JNGE)

Propósito: Salto condicional, e o sinal é ocupado.

Sintaxe:

JL símbolo

A instrução salta se for menor que, se não for maior que ou se for igual.

O salto é feito se SF for diferente de OF.

INSTRUÇÃO JLE (JNG)

Propósito: Salto condicional, e o sinal é ocupado.

Sintaxe:

JLE símbolo

A instrução salta se for menor que, se for igual ou se não for maior.

O salto é feito se ZF = 1 ou se SF for diferente de OF.

INSTRUÇÃO JC

Propósito: Salto condicional, e os flags são ocupados.

Sintaxe:

JC símbolo

A instrução salta se houver carry.

O salto é feito se CF = 1.

INSTRUÇÃO JNC

Propósito: Salto condicional, e o estado dos flags é ocupado.

Sintaxe:

JNC símbolo

A instrução salta se não houver carry.

O salto é feito se CF = 0.

INSTRUÇÃO JNO

Propósito: Salto condicional, e o estado dos flags é ocupado.

Sintaxe:

JNO símbolo

A instrução salta se não há overflow

O salto é feito se OF = 0.

INSTRUÇÃO JNP (JPO)

Propósito: Salto condicional, e o estado dos flags é ocupado.

Sintaxe:

JNP símbolo

A instrução salta se não houver paridade ou se a paridade for ímpar.

O salto é feito se PF = 0.

INSTRUÇÃO JNS

Propósito: Salto condicional, e o estado dos flags é ocupado.

Sintaxe:

JNP símbolo

A instrução salta se o sinal está desativado.

O salto é feito se SF = 0.

INSTRUÇÃO JO

Propósito: Salto condicional, e o estado dos flags é ocupado.

Sintaxe:

JO símbolo

A instrução salta se houver overflow.

O salto é feito se OF = 1.

INSTRUÇÃO JP (JPE)

Propósito: Salto condicional, e o estado dos flags é ocupado.

Sintaxe:

JP símbolo

A instrução salta se houver paridade ou se a paridade é par.

O salto é feito se PF = 1.

INSTRUÇÃO JS

Propósito: Salto condicional, e o estado dos flags é ocupado.

Sintaxe:

JS símbolo

A instrução salta se o sinal for ativo.

O salto é feito se SF =1.

Instruções para laços: LOOP

Estas instruções transferem a execução do processo, condicional ou incondicionalmente, para um destino, repetindo a ação até o contador ser zero.

LOOP

LOOPE

LOOPNE

INSTRUÇÃO LOOP

Propósito: Gerar um laço no programa, fazendo com que haja uma repetição.

Sintaxe:

LOOP símbolo

A instrução LOOP decrementa CX de um em um e transfere a execução do programa para o símbolo que é dado como operador, caso CX ainda não seja 1.

INSTRUÇÃO LOOPE

Propósito: Gerar um laço no programa, considerando o estado de ZF.

Sintaxe:

LOOPE símbolo

Esta instrução decrementa CX de um em um. Se CX for diferente de zero e ZF for igual a 1, então a execução do programa é transferida para o símbolo indicado como operador.

INSTRUÇÃO LOOPNE

Propósito: Gerar um laço no programa, considerando o estado de ZF.

Sintaxe:

LOOPNE símbolo

Esta instrução decrementa CX de um em um e transfere a execução do programa apenas se ZF for diferente de 0.

Instruções contadoras

Estas instruções são usadas para decrementar ou incrementar o conteúdo de contadores.

DEC

INC

INSTRUÇÂO DEC

Propósito: Decrementar o operador.

Sintaxe:

DEC destino

Esta instrução subtrai 1 do operador destino e armazena o novo valor no mesmo operador.

INSTRUÇÃO INC

Propósito: Incrementar o operador.

Sintaxe:

INC destino

Esta instrução adiciona 1 ao operador destino e mantém o resultado no mesmo operador.

Instruções de comparação

Estas instruções são usadas para comparar os operadores, e elas afetam o conteúdo dos flags.

CMP

CMPS (CMPSB) (CMPSW)

INSTRUÇÃO CMP

Propósito: Comparar os operadores.

Sintaxe:

CMP destino,fonte

Esta instrução subtrai o operador fonte do destino, mas não armazena o resultado da operação, apenas afeta o estado dos flags.

INSTRUÇÃO CMPS (CMPSB) (CMPSW)

Propósito: Comparar cadeias de um byte ou uma palavra.

Sintaxe:

CMP destino,fonte

Esta instrução compara efetuando uma subtração entre o byte ou palavra endereçado por DI, dentro do segmento extra de dados, e o byte ou palavra endereçado por SI dentro do segmento de dados, afetando o registador de flags, mas sem devolver o resultado da subtração. A instrução automaticamente incrementa ou decrementa os registadores de índice SI e DI, dependendo do valor do flag DF, de modo a indicar os próximos dois elementos a serem comparados. O valor de incremento ou decremento é uma de uma ou duas unidades, dependendo da natureza da operação.

Diante desta instrução, pode-se usar um prefixo para repetição, de modo a comparar dois blocos de memória entre si, repetindo a instrução de comparação até que ambos se tornem iguais ou desiguais.

Instruções de flag

Estas instruções afetam diretamente o conteúdo dos flags.

CLC

CLD

CLI

CMC

STC

STD

STI

INSTRUÇÃO CLC

Propósito: Limpar o flag de carry.

Sintaxe:

CLC

Esta instrução desliga o bit correspondente ao flag de carry. Por outras palavras, ajusta-o para zero.

INSTRUÇÃO CLD

Propósito: Limpar o flag de endereço.

Sintaxe:

CLD

Esta instrução desliga o bit correspondente ao flag de endereço.

INSTRUÇÃO CLI

Propósito: Limpar o flag de interrupção.

Sintaxe:

CLI

Esta instrução desliga o flag de interrupções, desabilitando, deste modo, interrupções mascaráveis.

Uma interrupção mascarável é aquela cujas funções são desativadas quando IF=0.

INSTRUÇÃO CMC

Propósito: Complementar o flag de carry.

Sintaxe:

CMC

Esta instrução complementa o estado do flag CF. Se CF = 0 a instrução o iguala a 1. Se CF = 1, a instrução o iguala a 0.

Poderíamos dizer que ela apenas inverte o valor do flag.

INSTRUÇÃO STC

Propósito: Ativar o flag de carry.

Sintaxe:

STC

Esta instrução ajusta para 1 o flag CF.

INSTRUÇÃO STD

Propósito: Ativar o flag de endereço.

Sintaxe:

STD

Esta instrução ajusta para 1 o flag DF.

INSTRUÇÃO STI

Propósito: Ativar o flag de interrupção.

Sintaxe:

STI

Esta instrução ativa o flag IF, e habilita interrupções externas mascaráveis (que só funcionam quando IF = 1).

Questionário

1. Complete a seguinte tabela sobre instruções de transferência e de carga.

INSTRUÇÃO	FUNÇÃO
INSTRUÇÃO MOV	
	Propósito: Mover byte ou cadeias de palavra da fonte, endereçada por SI, para o destino endereçado por DI.
INSTRUÇÕES LODS (LODSB) (LODSW)	
INSTRUÇÃO LDS	
	Propósito: Carregar o registador de segmento extra

2. Diga quais são as instruções que permitem usar a pilha para armazenar e recuperar dados.

3. Qual é a função da instrução NOT e AND?

4. Existem diversas nstruções aritméticas. Enuncia pelo menos 2 explicando a função das mesmas.

5. Como vimos ao longo do módulo existem diversas instruções de salto, relativamente às instruções JB e JC diga qual é a sintaxe de cada uma.

6. Qual é o principal objetivo de instruções contadoras?

Interrupções e Gestão de Ficheiros

Interrupções

Interrupções de hardware interno

Interrupções internas são geradas por certos eventos que ocorrem durante a execução de um programa. Este tipo de interrupções são geridas, na sua totalidade, pelo hardware e não é possível modificá-las.

Um exemplo claro deste tipo de interrupções é a que atualiza o contador do clock interno do computador, o hardware chama esta interrupção muitas vezes durante um segundo. Não nos é permitido gerir diretamente esta interrupção, uma vez que não se pode controlar a hora atualizada por software. Mas podemos usar os seus efeitos no computador para o nosso benefício, por exemplo para criar um virtual clock atualizado continuamente pelo contador interno de clock. Para tal, precisamos apenas ler o valor atual do contador e transforma-lo num formato compreensível pelo utilizador.

Interrupções de hardware externo

Interrupções externas são geradas através de dispositivos periféricos, tais como teclados, impressoras, placas de comunicação, entre outros. São também geradas por coprocessadores.

Não é possível desativar interrupções externas.

Estas interrupções não são enviadas diretamente para a CPU, mas, de uma forma melhor, são enviadas para um circuito integrado cuja função exclusiva é manusear este tipo de interrupção. O circuito em questão é o PIC8259A, é controlado pela CPU através de uma série de comunicação chamada paths.

Interrupções de software

Interrupções de software podem ser ativadas de maneira diferente pelos nossos programas assembly, invocando o número da interrupção desejada com a instrução INT.

O uso das interrupções facilita muito a criação dos programas, torna-os menores. Além disso, é fácil compreendê-las e dão uma boa performance.

Este tipo de interrupções podem ser separadas em duas categorias:

- Interrupções do Sistema Operacional DOS
- Interrupções do BIOS.

A diferença entre ambas é que as interrupções do sistema operacional são mais fáceis de usar, mas também são mais lentas, uma vez que acedam aos serviços do BIOS. Por outro lado, interrupções do BIOS são muito mais rápidas, mas possuem a desvantagem de serem parte do hardware, o que significa serem específicas à arquitetura do computador em questão.

A escolha sobre qual o tipo de interrupção usar irá depender somente das características que desejarmos dar ao nosso programa: velocidade (usamos BIOS), portabilidade (usamos DOS).

Interrupções mais comuns

Interrupção 21H

Propósito: Chamar uma diversidade de funções DOS.

Sintaxe:

Int 21H

Nota: Quando trabalhamos com o programa TASM é necessário especificar que o valor que estamos a usar está em hexadecimal.

Esta interrupção tem muitas funções, para aceder a cada uma delas é necessário que o número correspondente da função esteja no registador AH no momento da chamada da interrupção.

Funções para mostrar informações no ecrã.

02H Exibe um caracter

09H Exibe uma cadeia de caracteres

40H Escreve num dispositivo/ficheiro

Funções para ler informações do teclado.

01H Entrada do teclado

0AH Entrada do teclado usando buffer

3FH Leitura de um dispositivo/ficheiro

Funções para trabalhar com ficheiros.

Nesta seção são apenas especificadas as tarefas de cada função, para uma referência acerca dos conceitos usados.

Método FCB (blocos de controlo de ficheiro)

0FH Abertura de ficheiro

14H Leitura sequencial

15H Escrita sequencial

16H Criação de ficheiro

21H Leitura aleatória

22H Escrita aleatória

Handles (canais de comunicação)

3CH Criação de ficheiro

3DH Abertura de ficheiro

3EH Fechamento de ficheiro

3FH Leitura de ficheiro /dispositivo

40H Escrita de ficheiro /dispositivo

42H Move ponteiro de leitura/escrita num ficheiro

FUNÇÃO 02H

Propósito: Mostra um caracter no ecrã.

Registadores de chamada:

AH = 02H

DL = Valor de caracter a ser mostrado.

Registadores de retorno:

Nenhum.

Esta função mostra o caracter cujo código hexadecimal corresponde ao valor armazenado no registador DL, e não modifica nenhum registador.

O uso da função 40H é recomendado ao invés desta função.

FUNÇÃO 09H

Propósito: Mostra uma cadeia de caracteres no ecrã.

Registadores de chamada:

AH = 09H

DS:DX = Endereço de início da cadeia de caracteres.

Registadores de retorno:

Nenhum.

Esta função mostra os caracteres, um por um, a partir do endereço indicado nos registadores DS:DX até encontrar um caracter $, que é interpretado como fim da cadeia.

É recomendado usar a função 40H ao invés desta.

FUNÇÃO 40H

Propósito: Escrever num dispositivo ou num ficheiro.

Registadores de chamada:

AH = 40H

BX = Número do handle

CX = Quantidade de bytes a gravar

DS:DX = Área onde está o dado

Registadores de retorno:

CF = 0 se não houve erro

AX = Número de bytes escrito

CF = 1 se houve erro

AX = Código de erro

Para usar esta função para mostrar a informação no ecrã, faça o registador BX ser igual a 1, que é o valor default para o ecrã no DOS.

FUNÇÃO 01H

Propósito: Ler um caracter do teclado e mostrá-lo.

Registadores de chamada

AH = 01H

Registadores de retorno:

AL = Caracter lido

É muito fácil ler um caracter do teclado com esta função, o código hexadecimal do caracter lido é armazenado no registador AL. Nos caso de teclas especiais, como as de função F1, F2, além de outras, o registador AL conterá o valor 1, sendo necessário chamar a função novamente para obter o código daquele caracter.

FUNÇÃO 0AH

Propósito: Ler caracteres do teclado e armazená-los num buffer.

Registadores de chamada:

AH = 0AH

DS:DX = Endereço inicial da área de armazenamento

BYTE 0 = Quantidade de bytes na área

BYTE 1 = Quantidade de bytes lidos do BYTE 2 até BYTE 0 + 2 = caracteres lidos

Registadores de retorno:

Nenhum.

Os caracteres são lidos e armazenados num espaço de memória que foi definido. A estrutura deste espaço indica que o primeiro byte representará a quantidade máxima de caracteres que pode ser lida. O segundo, a quantidade de caracteres lidos e, no terceiro byte, o inicio onde eles são armazenados.

Quando se atinge a quantidade máxima permitida, ouve-se o som do speaker e qualquer carater adicional é ignorado.

Para finalizar a entrada, basta escrever [ENTER].

FUNÇÃO 3FH

Propósito: Ler informação de um dispositivo ou de um ficheiro.

Registadores de chamada:

AH = 3FH

BX = Número do handle

CX = Número de bytes a ler

DS:DX = Área para receber o dado

Registadores de retorno:

CF = 0 se não há erro e AX = número de bytes lidos.

CF = 1 se há erro e AX conterá o código de erro.

FUNÇÃO 0FH

Propósito: Abrir um ficheiro FCB.

Registadores de chamada:

AH = 0FH

DS:DX = Ponteiro para um FCB

Registadores de retorno:

AL = 00H se não há problemas, de outra forma retorna 0FFH

FUNÇÃO 14H

Propósito: Leitura sequencial num ficheiro FCB.

Registadores de chamada:

AH = 14H

DS:DX = Ponteiro para um FCB já aberto.

Registadores de retorno:

AL = 0 se não há erros, de outra forma o código correspondente de erro retornará:

1 - erro no fim do ficheiro

2 - erro na estrutura FCB

3 - erros de leitura parcial.

O que esta função faz é ler o próximo bloco de informações do endereço dado por DS:DX, e atualizar este registo.

FUNÇÃO 15H

Propósito: Escrita sequencial e arquivo FCB.

Registadores de chamada:

AH = 15H

DS:DX = Ponteiro para um FCB já aberto.

Registadores de retorno:

AL = 00H se não há erros, de outra forma conterá o código de erro:

1 - disco cheio ou arquivo somente de leitura

2 - erro na formação ou na especificação do FCB.

A função 15H atualiza o FCB após a escrita do registo para o presente bloco.

FUNÇÃO 16H

Propósito: Criar um ficheiro FCB.

Registadores de chamada:

AH = 16H

DS:DX = Ponteiro para um FCB já aberto.

Registadores de retorno:

AL = 00H se não há erros, de outra forma conterá o valor 0FFH.

É baseada na informação advinda de um FCB para criar um ficheiro num disco.

FUNÇÃO 21H

Propósito: Ler de modo aleatório um ficheiro FCB.

Registadores de chamada:

AH = 21H

DS:DX = Ponteiro para FCB aberto.

Registadores de retorno:

A = 00H se não há erro, de outra forma AH conterá o código de erro:

1 - se é o fim do ficheiro

2 - se há um erro de especificação no FCB

3 - se um registo foi lido parcialmente ou o ponteiro do ficheiro está no fim do mesmo.

Esta função lê o registo especificado pelos campos do bloco atual e registo de um FCB aberto e coloca a informação na DTA, Área de Transferência do Disco.

FUNÇÃO 22H

Propósito: Escrita aleatória num ficheiro FCB.

Registadores de chamada:

AH = 22H

DS:DX = Ponteiro para um FCB aberto.

Registadores de retorno:

AL = 00H se não há erro, de outra forma conterá o código de erro:

1 - se o disco está cheio ou o ficheiro é apenas de leitura

2 - se há um erro na especificação FCB.

Escreve o registo especificado pelos campos do bloco atual e registo de um FCB aberto. Esta informação é do conteúdo da DTA.

FUNÇÃO 3CH

Propósito: Criar um ficheiro se não existir ou deixá-lo com comprimento 0 se existir.

Registadores de chamada:

AH = 3CH

CH = Atributo do ficheiro

DS:DX = Nome do ficheiro, no formato ASCII.

Registadores de retorno:

CF = 0 e AX informa o número do handle se não há erro. Se houver erro, CF será 1 e AX conterá o código de erro:

3 - caminho não encontrado,

4 - não há handles disponíveis

5 - acesso negado.

Esta função substitui a função 16H. O nome do ficheiro é especificado numa cadeia ASCII de bytes terminados pelo caracter 0.

O ficheiro criado conterá os atributos definidos no registador CX, do seguinte modo:

Valor	**Atributos**
00H	Normal
02H	Hidden
04H	System
06H	Hidden e System

O ficheiro é criado com permissão de leitura e escrita. Não é possível a criação de diretórios através desta função.

FUNÇÃO 3DH

Propósito: Abre um ficheiro e retorna um handle.

Registadores de chamada:

AH = 3DH

AL = modo de acesso

DS:DX = Nome do ficheiro, no formato ASCII.

Registadores de retorno:

CF = 0 e AX = número do handle se não houver erros, de outra forma CF = 1 e AX = código de erro:

01H - se a função não é válida

02H - se o ficheiro não foi encontrado

03H - se o caminho não foi encontrado

04H - se não há handles disponíveis

05H - acesso negado

0CH - se o código de acesso não é válido.

O handle retornado é de 16 bits.

O código de acesso é especificado da seguinte maneira:

BITS

7 6 5 4 3 2 1

. . . . 0 0 0 Apenas leitura

. . . . 0 0 1 Apenas escrita

. . . . 0 1 0 Leitura/Escrita

. . . x . . . RESERVADO

FUNÇÃO 3EH

Propósito: Fecha um ficheiro (handle).

Registadores de chamada:

AH = 3EH

BX = Número do handle associado

Registadores de retorno:

CF = 0 se não houver erros, ou CF será 1 e AX conterá o código de erro:

06H - se o handle for inválido.

Esta função atualiza o ficheiro e libera o handle que estava a utilizar.

FUNÇÃO 3EH

Propósito: Fecha um ficheiro (handle).

Registadores de chamada:

AH = 3EH

BX = Número do handle associado

Registadores de retorno:

CF = 0 se não houver erros, ou CF será 1 e AX conterá o código de erro:

06H - se o handle é inválido.

Esta função atualiza o ficheiro e libera o handle que estava a utilizar.

FUNÇÃO 3FH

Propósito: Ler uma quantidade específica de bytes de um ficheiro aberto e armazená-los num buffer específico.

Interrupção 10h

Propósito: Chamar uma diversidade de funções do BIOS

Sintaxe:

Int 10H

Esta interrupção tem várias funções, todas para entrada e saída de vídeo.

Para aceder a cada uma delas é necessário colocar o número da função correspondente no registador AH.

Veremos apenas as funções mais comuns da interrupção 10H.

- Função 02H, seleciona a posição do cursor
- Função 09H, exibe um caracter e o atributo na posição do cursor
- Função 0AH, exibe um caracter na posição do cursor
- Função 0EH, modo alfanumérico de exibição de caracteres

Função 02h

Propósito: Move o cursor no ecrã do computador usando o modo texto.

Registadores de chamada:

AH = 02H

BH = Página de vídeo onde o cursor está posicionado.

DH = linha

DL = coluna

Registadores de retorno:

Nenhum.

A posição do cursor é definida pelas suas coordenadas, iniciando-se na posição 0,0 até a posição 79,24. Logo os valores possíveis para os registadores DH e DL são: de 0 a 24 para linhas e de 0 a 79 para colunas.

Função 09h

Propósito: Mostra um determinado caracter várias vezes no ecrã do computador com um atributo definido, iniciando pela posição atual do cursor.

Registadores de chamada:

AH = 09H

AL = Caracter a exibir

BH = Página de vídeo, onde o caracter será mostrado

BL = Atributo do caracter

CX = Número de repetições.

Registadores de retorno:

Nenhum

Esta função mostra um caracter no ecrã várias vezes, de acordo com o número especificado no registador CX, mas sem mudar a posição do cursor no ecrã.

Função 0Ah

Propósito: Exibe um caracter na posição atual do cursor.

Registadores de chamada:

AH = 0AH

AL = Caracter a exibir

BH = Página de vídeo onde o caracter será exibido

BL = Cor do caracter (apenas em modo gráfico)

CX = Número de repetições

Registadores de retorno:

Nenhum.

A principal diferença entre esta função e a anterior é permitir mudança nos atributos, bem como mudar a posição do cursor.

Função 0EH

Propósito: Exibir um caracter no ecrã do computador atualizando a posição do cursor.

Registadores de chamada:

AH = 0EH

AL = Caracter a exibir

BH = Página de vídeo onde o caracter será exibido

BL = Cor a usar (apenas em modo gráfico)

Registadores de retorno:

Nenhum

Interrupção 16H

Veremos duas funções da interrupção 16H. A exemplo das restantes interrupções, usa-se o registador AH para chamá-las.

- Funções da interrupção 16h
- Função 00H, lê um caracter do teclado.
- Função 01H, lê o estado atual do teclado.

Função 00H

Propósito: Ler um caracter do teclado.

Registadores de chamada:

AH = 00H

Registadores de retorno:

AH = Código da tecla pressionada

AL = Valor ASCII do caracter

Quando se usa esta interrupção, os programas executam até que uma tecla seja pressionada. Se for um valor ASCII, é armazenado no registador AH. Caso contrário, o código é armazenado no registador AL e AH=0.

Este valor de AL pode ser utilizado quando queremos detetar teclas que não estão diretamente representadas pelo seu valor ASCII, tais como [ALT][CONTROL].

Função 01h

Propósito: Ler o estado do teclado

Registadores de chamada:

AH = 01H

Registadores de retorno:

Se o registador de flag é zero, significa que há informação no buffer de teclado na memória. Caso contrário, o buffer está vazio. Portanto o valor do registador AH será o valor da tecla armazenada no buffer.

Interrupção 17H

Propósito: Manusear a entrada e saída da impressora.

Sintaxe:

Int 17H

Esta interrupção é usada para enviar carateres, fazer o set ou ler o estado de uma impressora.

Funções da interrupção 17h

- Função 00H, imprime um valor ASCII
- Função 01H, faz um set à impressora
- Função 02H, lê estado da impressora

Função 00H

Propósito: Imprimir um caracter numa impressora.

Registadores de chamada:

AH = 00H

AL = Carater a imprimir

DX = Porta de conexão

Registadores de retorno:

AH = Estado da impressora

Os valores da porta a colocar no registador DX são:

LPT1 = 0, LPT2 = 1, LPT3 = 2 ...

O estado da impressora é codificado bit a bit como segue:

BIT 1/0 SIGNIFICADO

---

0 - 1 Estado de time-out

1 - ----------

2 - ----------

3 - 1 Erro de entrada e saída

4 - 1 Impressora selecionada

5 - 1 Fim de papel

6 - 1 Reconhecimento de comunicação

7 - 1 A impressora está pronta para o uso

Os bits 1 e 2 não são relevantes

A maioria dos BIOS suportam 3 portas paralelas, havendo alguns que suportam 4.

Função 01h

Propósito: Fazer um Set numa porta paralela.

Registadores de chamada:

AH = 01H

DX = Porta

Registadores de retorno:

AH = Status da impressora

A porta definida no registador DX pode ser: LPT1=0, LPT2=1, e assim por diante.

O estado da impressora é codificado bit a bit como segue:

BIT 1/0 SIGNIFICADO

---

0 - 1 Estado de time-out

1 - ----------

2 - ----------

3 - 1 Erro de entrada e saída

4 - 1 Impressora selecionada

5 - 1 Fim de papel

6 - 1 Reconhecimento de comunicação

7 - 1 A impressora está pronta para o uso

Os bits 1 e 2 não são relevantes.

Função 02h

Propósito: Obter o status da impressora.

Registadores de chamada:

AH = 01H

DX = Porta

Registadores de retorno

AH = Status da impressora

A porta definida no registador DX pode ser: LPT1=0, LPT2=1, e assim por diante.

O estado da impressora é codificado bit a bit como se segue:

BIT 1/0 SIGNIFICADO

---

0 - 1 Estado de time-out

1 - ----------

2 - ----------

3 - 1 Erro de entrada e saída

4 - 1 Impressora selecionada

5 - 1 Fim de papel

6 - 1 Reconhecimento de comunicação

7 - 1 A impressora está pronta para utilizar

Os bits 1 e 2 não são relevantes.

Gestão de ficheiros

Modos de trabalhar com ficheiros

Há dois modos de trabalhar com ficheiros. O primeiro é através de FCB (blocos de controlo de ficheiro), o segundo é através de canais de comunicação, também conhecidos como handles.

O primeiro modo de manusear ficheiros tem sido usado desde o sistema operacional CPM, predecessor do DOS, logo permite certas compatibilidades com muitos ficheiros velhos do CPM bem como com a versão 1.0 do DOS, além deste método permitir-nos ter um número ilimitado de ficheiros abertos ao mesmo tempo. Se quisermos criar uma partição de um disco, a única forma é através deste método.

Depois de considerarmos as vantagens de FCB, o uso do método de Canais de Comunicação é muito simples e permite-nos um melhor manuseio de erros.

Para uma melhor facilidade, daqui por diante vamos referir-nos aos Blocos de Controlo de ficheiro como FCBs e aos Canais de Comunicação como handles.

Método FCB

Introdução

Há dois tipos de FCB, o normal, cujo comprimento é 37 bytes, e o estendido, com 44 bytes.

O FCB é composto de informações dadas pelo programador e por informações que ele obtém diretamente do sistema operacional. Quando estes tipos de ficheiros são usados, só é possível se trabalhar na pasta atual, pois FCBs não fornecem suporte ao sistema de organização de ficheiros através de pastas do DOS.

FCB é composto pelos seguintes campos:

POSIÇÃO	COMPRIMENTO	SIGNIFICADO
00H	1 Byte	Drive
01H	8 Bytes	Nome do ficheiro
09H	3 Bytes	Extensão
0CH	2 Bytes	Número do bloco

0EH	2 Bytes	Tamanho do registo
10H	4 Bytes	Tamanho do ficheiro
14H	2 Bytes	Data de criação
16H	2 Bytes	Hora de criação
18H	8 Bytes	Reservado
20H	1 Bytes	Registo atual
21H	4 Bytes	Registro aleatório

Para selecionar o drive de trabalho, assumamos:

drive A = 1;

drive B = 2; etc.

Se for usado 0, o drive que está a ser usado no momento será obtido como opção.

O nome do ficheiro deve ser justificado à esquerda e é necessário preencher com espaços os bytes remanescentes, a extensão é colocada do mesmo modo.

O bloco atual e o registo atual dizem ao computador que registo será acedido nas operações de leitura e escrita. Um bloco é um grupo de 128 registos. O primeiro bloco de ficheiros é o bloco 0. O primeiro registo é o registo 0, logo o último registo do primeiro bloco deve ser o 127, uma vez que a numeração é iniciada com 0 e o bloco pode conter 128 registadores no total.

Abertura de ficheiros

Para abrir um ficheiro FCB é usada a função 0FH da interrupção 21h (ver na secção de interrupções).

A unidade, o nome e a extensão do ficheiro devem ser inicializadas antes da abertura.

O registador DX deve apontar para o bloco. Se o valor FFH for retornado no registador AH quando da chamada da interrupção, então o ficheiro não foi encontrado. Se tudo der certo, o valor 0 é retornado.

Se o ficheiro é aberto, então o DOS inicializa o bloco corrente em 0, o tamanho do registo para 128 bytes. O tamanho do ficheiro e a sua data são preenchidos com as informações encontradas na pasta.

Criar um novo ficheiro

Para a criação de ficheiros é usada a função 16H da interrupção 21h.

O registador DX deve apontar para uma estrutura de controlo cujos requisitos são de que pelo menos a unidade lógica, o nome e a extensão do ficheiro sejam definidas.

Caso ocorra problema, o valor FFH deve retornar em AL, de outra forma este registador conterá o valor 0.

Escrita sequencial

Antes de conseguirmos realizar escrita para o disco, é necessário definir a área de transferência de dados usando, para tal, a função 1AH da interrupção 21h.

A função 1AH não retorna qualquer estado do disco nem da operação. Mas a função 15H, que usaremos para escrever para o disco, faz isso no registador AL. Se este for igual a zero, então não há erro e os campos de registo atual e de bloco são atualizados.

Leitura sequencial

Antes de tudo, devemos definir a área de transferência de arquivo ou DTA.

Para a leitura sequencial usaremos a função 14H da interrupção 21h.

O registro a ser lido é definido pelos campos registro e bloco atual. O registador AL retorna o estado da operação.

Se AL contém o valor 1 ou 3, significa que foi atingido o fim do arquivo. Um valor 2, por sua vez, significa que o FCB está estruturado erroneamente.

Caso não ocorra erro, AL conterá o valor 0 e os campos de registro e bloco atual são atualizados.

Leitura e escrita aleatória

A função 21H e a função 22H da interrupção 21h são usadas à realização, respetivamente, da escrita e leitura aleatória.

O número de registro randômico e o bloco corrente são usados para calcular a posição relativa do registro a ser lido ou escrito.

O registador AL retorna a mesma informação do que par a escrita e leitura sequencial. A informação a ser lida será retornada na área de transferência do disco, bem como a informação a ser escrita retorna na DTA.

Fechar um ficheiro

Para fechar um ficheiro usamos a função 10H da interrupção 21h.
Se após invocar esta função, o registador AL contiver o valor FFH, significa que o ficheiro foi mudado de posição, o disco foi mudado ou há erro de acesso ao disco.

Canais de comunicação

Trabalhar com handles

O uso de handles para gerir ficheiros traz grandes facilidades na criação de ficheiros e o programador pode concentrar-se em outros aspetos da programação sem se preocupar com detalhes que podem ser manuseados pelo sistema operacional.
A facilidade dos handles consiste em que para operarmos sobre um ficheiro é apenas necessário definirmos o nome do mesmo e o número de handle a usar, a restante informação é manuseada internamente pelo DOS.
Quando usamos este método para trabalhar com ficheiros, não há distinção entre acesso sequencial ou aleatório, o ficheiro é simplesmente usado como uma rede de bytes.

Funções para usar Handles

As funções usadas para o manuseio de ficheiros através de handles estão descritas no subcapítulo: Interrupções, na seção dedicada à interrupção 21h.

Questionário

1. As interrupções podem ser de vários tipos, interrupções de hardware interno, hardware externo e de software. Diga o que entende dobre interrupções de hardware internas.

2. Quais são as interrupções mais comuns?

3. Existem dois modos para trabalharmos com ficheiros. Quais?

4. Como é feita a abertura de um ficheiro FCB e o seu encerramento?

5. Qual é uma das principais vantagens na utilização de handles?

Macros e Procedimentos

Procedimentos

Definição de um procedimento

Um procedimento é uma coleção de instruções para as quais é possível direcionar o curso de nosso programa, e uma vez que a execução destas instruções do procedimento tenha acabado, o controlo retorna para linha que segue à que chamou o procedimento. Procedimentos ajudam-nos a criar programas legíveis e fáceis de modificar.
Quando se invoca um procedimento, o endereço da próxima instrução do programa é mantido na pilha, de onde é recuperado quando do retorno do procedimento.

Sintaxe de um procedimento

Há dois tipos de procedimentos, os intrasegments, que se localizam no mesmo segmento da instrução que o chama, e os intersegments, que podem localizar-se em diferentes segmentos de memória.
Quando os procedimentos intrasegments são usados, o valor de IP é armazenado na pilha e quando os procedimentos intersegments são usados o valor de CS:IP é armazenado. Lembre-se que o registador CS indica qual o segmento de código.
A diretiva que chama um procedimento é:

```
CALL NomedoProcedimento
```

As partes que compõem um procedimento são as seguintes:

- Declaração do procedimento
- Código do procedimento
- Diretiva de retorno
- Término do procedimento

Por exemplo, se quisermos uma rotina que some dois bytes armazenados em AH e AL, e o resultado da soma em BX:

```
Soma Proc Near        ; Declaração do Procedimento
        Mov BX, 0     ; Conteúdo do Procedimento...
```

```
        Mov BL, AH
        Mov AH, 00
        Add BX, AX
        Ret             ; Diretiva de retorno
Soma EndP               ; Fim do Procedimento
```

Na declaração, a primeira palavra, Soma, corresponde ao nome do procedimento. Proc declara-o e a palavra Near indica que o procedimento é do tipo intrasegment, ou seja, no mesmo segmento. A diretiva Ret carrega IP com o endereço armazenado na pilha para retornar ao programa que chamou. Finalmente, Soma EndP indica o fim do procedimento.

Para declarar um procedimento inter segment, basta substituir a palavra Near para FAR. A chamada deste procedimento é feito de modo idêntico:

```
Call Soma
```

Macros oferecem uma grande flexibilidade na programação, comparadas aos procedimentos.

Macros

Definição de uma Macro

Uma macro é um grupo de instruções repetitivas num programa que são codificadas apenas uma vez e, assim, poupam espaço, podendo ser utilizadas tantas vezes quantas forem necessário.

A principal diferença entre uma macro e um procedimento é que numa macro é possível a passagem de parâmetros e num procedimento não. No momento em que a macro é executada, cada parâmetro é substituído pelo nome ou valor especificado na hora da chamada.

Podemos dizer, desta forma, que um procedimento é uma extensão de um determinado programa, enquanto que uma macro é um módulo que especifica funções que podem ser utilizadas por diferentes programas.

Uma outra diferença entre uma macro e um procedimento é o modo de chamada de cada um. Para chamar um procedimento, é necessário a diretiva CALL, por outro lado, para chamada de macros é feita com se fosse uma instrução normal da linguagem assembly.

Sintaxe de uma Macro

As partes que compõem uma macro são as seguintes:

- Declaração da macro
- Código da macro
- Diretiva de término da macro

A declaração da macro é feita como se segue:

```
NomeMacro MACRO [parâmetro1, parâmetro2...]
```

Do mesmo modo que temos a funcionalidade dos parâmetros, é possível também a criação de uma macro que não os possua.

A diretiva de término da macro é: ENDM

Um exemplo de uma macro para colocar o cursor numa determinada posição do ecrã:

```
Pos MACRO Linha, Coluna
        PUSH AX
        PUSH BX
        PUSH DX
        MOV AH, 02H
        MOV DH, Linha
        MOV DL, Coluna
        MOV BH, 0
        INT 10H
        POP DX
        POP BX
        POP AX
ENDM
```

Para usar uma macro basta chamá-la pelo seu nome, tal como se fosse qualquer instrução na linguagem assembly:

```
Pos 8, 6
```

Biblioteca de Macros

Uma das facilidades oferecidas pelo uso de macros é a criação de bibliotecas, que são grupo de macros, podendo ser incluídas num programa originárias de ficheiros diferentes. A criação destas bibliotecas é muito simples. Criamos um ficheiro com todas as macros que serão necessárias e guardamo-lo como um ficheiro de texto.

Para incluir uma biblioteca num programa, basta colocar a seguinte instrução:

Include NomedoFicheiro na parte inicial do programa, antes da declaração do modelo de memória.

Supondo que o ficheiro de macros tenha sido guardado com o nome de MACROS.TXT, a instrução Include seria utilizada do seguinte modo:

```
;Início do programa
Include MACROS.TXT
        .MODEL SMALL
        .DATA
        ;Os dados inserem-se aqui
        .CODE
        Inicio:
                ;O código do programa começa aqui
                .STACK
                ;A pilha é declarada
        End Inicio
;Fim do programa
```

Questionário

1. Diga o que entende por procedimento.

2. Como é feita a chamada de um procedimento?

3. Qual é a definição e Macro?

4. Em que diferem as Macros e os Procedimentos?

5. Qual é a diretiva para terminar um macro?

Exemplos de Programas

Exemplos de Programas com Debug

Serão apresentados a seguir alguns programas feitos no debug do DOS.

Podemos executar cada programa assembly usando o comando "g" (go), para ver o que cada programa faz.

Procedimento

1. *Carregar o programa exemplo*

Por exemplo:

```
C:\>debug
-n one.com
-1
-u 100 109
0D80:0100 B80600 MOV AX,0006
0D80:0103 BB0400 MOV BX,0004
0D80:0106 01D8 ADD AX,BX
0D80:0108 CD20 INT 20
-
```

Nota:

-n one.com

Dar nome ao programa a ser carregado

-1

Carregá-lo

-u 100 109

Desmontar o código do endereço inicial ao final especificado

Escreva o comando g

Por exemplo:

-g

O programa termina normalmente

EXEMPLOS DE PROGRAMAS NO DEBUG

1º Exemplo:

```
-a100
297D:0100    MOV AX,0006 ;Põe o valor 0006 no registador AX
297D:0103    MOV BX,0004       ;Põe o valor 0004 no registador BX
297D:0106    ADD AX,BX         ;Adiciona BX ao conteúdo de AX
297D:0108    INT 20            ;Finaliza o Programa
```

A única coisa que este programa faz é guardar dois valores em dois registadores e adicionar o valor de um ao outro.

2º Exemplo:

```
- a100
0C1B:0100    jmp 125           ;Salta para o endereço 125h
0C1B:0102    [Enter]
- e 102 'Olá, como estas?' 0d 0a '$'
- a125
0C1B:0125    MOV DX,0102 ;     Copia a string para registador DX
0C1B:0128    MOV CX,000F       ;Quantas vezes a string será apresentada
0C1B:012B    MOV AH,09         ;Copia o valor 09 para registador AH
0C1B:012D    INT 21            ;Mostra a string
0C1B:012F    DEC CX            ;Subtrai 1 de CX
0C1B:0130    JCXZ 0134         ;Se CX é igual a 0 salta para o endereço 0134
0C1B:0132    JMP 012D          ;Salta ao endereço 012D
0C1B:0134    INT 20            ;Finaliza o programa
```

Este programa mostra 15 vezes no ecrã a string de caracteres.

3º Exemplo:

```
-a100
297D:0100    MOV AH,01         ;Função para mudar o cursor
297D:0102    MOV CX,0007       ;Formata o cursor
297D:0105    INT 10            ;Chama interrupção do BIOS
297D:0107    INT 20            ;Finaliza o programa
```

Este programa muda o formato do cursor.

4º Exemplo:

```
-a100
297D:0100   MOV AH,01        ;Função 1 (lê caractere do teclado)
297D:0102   INT 21           ;Chama interrupção do DOS
297D:0104   CMP AL,0D        ;Compara se o caractere lido é um ENTER
297D:0106   JNZ 0100         ;Se não é, lê um outro caractere
297D:0108   MOV AH,02        ;Função 2 (escreve um caractere no ecrã)
297D:010A   MOV DL,AL        ;Character to write on AL
297D:010C   INT 21           ;Chama interrupção do DOS
297D:010E   INT 20           ;Finaliza o programa
```

Este programa usa a interrupção 21h do DOS. Usa duas funções da mesma: a primeira lê um caractere do teclado (função 1) e a segunda (função 2) escreve um caractere no ecrã. O programa lê caracteres do teclado até encontrar um ENTER.

5º Exemplo:

```
-a100
297D:0100   MOV AH,02        ;Função 2 (escreve um caractere no ecrã)
297D:0102   MOV CX,0008      ;Põe o valor 0008 no registador CX
297D:0105   MOV DL,00        ;Põe o valor 00 no registador DL
297D:0107   RCL BL,1         ;Rotação do byte em BL um bit para a esquerda
297D:0109   ADC DL,30        ;Converte o registador de flag para 1
297D:010C   INT 21           ;Chama interrupção do DOS
297D:010E   LOOP 0105        ;Salta se CX > 0 para o endereço 0105
297D:0110   INT 20           ;Finaliza o programa
```

Este programa mostra no ecrã um número binário através de um ciclo condicional (LOOP) usando a rotação do byte.

6º Exemplo:

```
-a100
297D:0100   MOV AH,02        ;Função 2 (escreve um caractere no ecrã)
297D:0102   MOV DL,BL        ;Põe o valor de BL em DL
297D:0104   ADD DL,30        ;Adiciona o valor 30 a DL
```

```
297D:0107    CMP DL,3A     ;Compara o valor 3A com o conteúdo de DL sem afetá-lo
                           ;O seu valor apenas modifica o estado do flag de carry
297D:010A    JL 010F       ;salta ao endereço 010f, se for menor

297D:010C    ADD DL,07     ;Adiciona o valor 07 a DL
297D:010F    INT 21        ;Chama interrupção do DOS
297D:0111    INT 20        ;Finaliza o programa
```

Este programa imprime um valor zero em dígitos hexadecimais.

7º Exemplo:

```
-a100
297D:0100    MOV AH,02     ;Função 2 (escreve um caractere no ecrã)
297D:0102    MOV DL,BL     ;Põe o valor de BL em DL
297D:0104    AND DL,0F     ;Transporta fazendo AND dos números bit a bit
297D:0107    ADD DL,30     ;Adiciona 30 a Dl
297D:010A    CMP DL,3A     ;Compara Dl com 3A
297D:010D    JL 0112       ;Salta ao endereço 0112, se menor
297D:010F    ADD DL,07     ;Adiciona 07 a DL
297D:0112    INT 21        ;Chama interrupção do DOS
297D:0114    INT 20        ;Finaliza o programa
```

Este programa é usado para imprimir dois dígitos hexadecimais.

8º Exemplo:

```
-a100
297D:0100    MOV AH,02     ;Função 2 (escreve um caractere no ecrã)
297D:0102    MOV DL,BL     ;Põe o valor de BL em DL
297D:0104    MOV CL,04     ;Põe o valor 04 em CL
297D:0106    SHR DL,CL     ;Desloca os 4 bits mais altos do número ao nibble
mais à direita
297D:0108    ADD DL,30     ;Adiciona 30 a DL
297D:010B    CMP DL,3A     ;Compara Dl com 3A
297D:010E    JL 0113       ;Salta ao endereço 0113, se menor
```


```
297D:0110   ADD DL,07          ;Adiciona 07 a DL
297D:0113   INT 21             ;Chama interrupção do DOS
297D:0115   INT 20             ;Finaliza o programa
```

Este programa imprime o primeiro de dois dígitos hexadecimais.

9º Exemplo:

```
-a100
297D:0100   MOV AH,02          ;Função 2 (escreve um caractere no ecrã)
297D:0102   MOV DL,BL          ;Põe o valor de BL em DL
297D:0104   MOV CL,04          ;Põe o valor 04 em CL
297D:0106   SHR DL,CL          ;Desloca os 4 bits mais altos do número ao nibble
mais à direita
297D:0108   ADD DL,30          ;Adiciona 30 a DL
297D:010B   CMP DL,3A          ;Compara Dl com 3A
297D:010E   JL 0113            ;Salta ao endereço 0113, se menor
297D:0110   ADD DL,07          ;Adiciona 07 a DL
297D:0113   INT 21             ;Chama interrupção do DOS
297D:0115   MOV DL,BL          ;Põe o valor de BL em DL
297D:0117   AND DL,0F          ;Transporta fazendo AND dos números bit a bit
297D:011A   ADD DL,30          ;Adiciona 30 a DL
297D:011D   CMP DL,3A          ;Compara Dl com 3A
297D:0120   JL 0125            ;Salta ao endereço 0125, se menor
297D:0122   ADD DL,07          ;Adiciona 07 a DL
297D:0125   INT 21             ;Chama interrupção do DOS
297D:0127   INT 20             ;Finaliza o programa
```

Este programa imprime o segundo de dois dígitos hexadecimais.

10º Exemplo

```
-a100
297D:0100   MOV AH,01          ;Função 1 (lê caractere do teclado)
297D:0102   INT 21             ;Chama interrupção do DOS
297D:0104   MOV DL,AL          ;Põe o valor de AL em DL
```

```
297D:0106   SUB DL,30       ;Subtrai 30 de DL
297D:0109   CMP DL,09       ;Compara DL com 09
297D:010C   JLE 0111        ;Salta ao endereço 0111, se menor ou igual
297D:010E   SUB DL,07       ;Subtrai 07 de DL
297D:0111   MOV CL,04       ;Põe o valor 04 em CL
297D:0113   SHL DL,CL       ;Insere zeros à direita
297D:0115   INT 21          ;Chama interrupção do DOS
297D:0117   SUB AL,30       ;Subtrai 30 de AL
297D:0119   CMP AL,09       ;Compara AL com 09
297D:011B   JLE 011F        ;Salta ao endereço 011f, se menor ou igual
297D:011D   SUB AL,07       ;Subtrai 07 de AL
297D:011F   ADD DL,AL       ;Adiciona AL a DL
297D:0121   INT 20          ;Finaliza o programa
```

Este programa pode ler dois dígitos hexadecimais.

11º Exemplo:

```
-a100
297D:0100   CALL 0200       ;Chama um procedimento
297D:0103   INT 20          ;Finaliza o programa
-a200
297D:0200   PUSH DX         ;Põe o valor de DX na pilha
297D:0201   MOV AH,08       ;Função 8
297D:0203   INT 21          ;Chama interrupção do DOS
297D:0205   CMP AL,30       ;Compara AL com 30
297D:0207   JB 0203         ;Salta se CF for ativado ao endereço 0203
297D:0209   CMP AL,46       ;Compara AL com 46
297D:020B   JA 0203         ;Salta ao endereço 0203, se diferente
297D:020D   CMP AL,39       ;Compara AL com 39
297D:020F   JA 021B         ;Salta ao endereço 021B, se diferente
297D:0211   MOV AH,02       ;Função 2 (escreve um caractere no ecrã)
297D:0213   MOV DL,AL       ;Põe o valor de AL em DL
297D:0215   INT 21          ;Chama interrupção do DOS
```


```
297D:0217   SUB AL,30          ;Subtrai 30 de AL
297D:0219   POP DX             ;Extrai o valor de DX da pilha
297D:021A   RET                ;Retorna o controlo ao programa principal
297D:021B   CMP AL,41          ;Compara AL com 41
297D:021D   JB 0203            ;Salta se CF é ativado ao endereço 0203
297D:021F   MOV AH,02          ;Função 2 (escreve um caractere no ecrã)
297D:022    MOV DL,AL          ;Põe o valor AL em DL
297D:0223   INT 21             ;Chama interrupção do DOS
297D:0225   SUB AL,37          ;Subtrai 37 de AL
297D:0227   POP DX             ;Extrai o valor de DX da pilha
297D:0228   RET                ;Retorna o controlo ao programa principal
```

Este programa mantém-se a ler caracteres até receber um que possa ser convertido para um número hexadecimal.

Exemplos de Programas com TASM

Nesta seção são apresentados vários exemplos de programas a serem montados fazendo uso do TASM da Borland.

Procedimento:

Para montá-los, siga os seguintes passos:

1. **Montar o programa**

Por exemplo:

```
C:\>tasm one.asm
Turbo Assembler Version 2.0 Copyright (c) 1988, 1990 Borland International
Assembling file: one.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 471k
C:\>
```

Isto criará um programa objeto com o mesmo nome do fonte, neste caso: one.obj

2. **Criar o programa executável**

Por exemplo:

C:\>tlink one.obj

Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International

C:\>

Isto cria o programa executável com o mesmo nome do objeto e com extensão diferente, one.exe

3. **Executar programa executável.**

Basta escrever o nome do programa criado.

EXEMPLOS DE PROGRAMAS ASSEMBLY

1º Exemplo:

```
;nome do programa: one.asm
;
.model small
.stack
.code
mov AH,1h      ;Função 1 do DOS
Int 21h        ;lê o caracter e retorna o código ASCII ao registador AL
mov DL,AL      ;move o código ASCII para o registador DL
sub DL,30h     ;subtrai de 30h para converter a um dígito de 0 a 9
cmp DL,9h      ;compara se o dígito está entre 0 e 9
jle digit1     ;se verdadeiro obtém o primeiro número (4 bits)
sub DL,7h      ;se falso, subtrai de 7h para converter a uma letra A-F
digit1:
        mov CL,4h      ;prepara para multiplicar por 16
        shl DL,CL      ;multiplica para converter dentro dos 4 bits mais altos
        int 21h        ;obtém o próximo caracter
        sub AL,30h     ;repete a operação de conversão
        cmp AL,9h      ;compara o valor 9h com o conteúdo do registador AL
        jle digit2     ;se verdadeiro, obtém o segundo dígito
```


```
        sub AL,7h       ;se falso, subtrai de 7h
digit2:
        add DL,AL       ;adiciona o segundo dígito
        mov AH,4Ch      ;função 4Ch do DOS (exit)
Int 21h         ;interrupção 21h
End             ;finaliza o programa
```

Este programa lê dois caracteres e os imprime na tela

2º Exemplo:

```
;nome do programa: two.asm
.model small
.stack
.code
PRINT_A_J PROC
MOV DL,'A'                  ;mover o caractere A para o registador DL
MOV CX,10                   ;move o valor decimal 10 para o registador CX
;este valor é usado para fazer laço com 10 interações
PRINT_LOOP:
        CALL WRITE_CHAR     ;Imprime o caracter em DL
        INC DL              ;Incrementa o valor do registador DL
        LOOP PRINT_LOOP     ;Laço para imprimir 10 caracteres
                MOV AH,4Ch  ;Função 4Ch, para sair ao DOS
        INT 21h             ;Interrupção 21h
PRINT_A_J ENDP              ;Finaliza o procedimento
WRITE_CHAR PROC
        MOV AH,2h           ;Função 2h, imprime caracter
        INT 21h             ;Imprime o caracter que está em DL
        RET                 ;Retorna o controlo ao procedimento que chamou
WRITE_CHAR ENDP             ;Finaliza o procedimento
END PRINT_A_J               ;Finaliza o programa
```

Este programa mostra os caracteres ABCDEFGHIJ na tela.

3º Exemplo:

```
;nome do programa: three.asm
.model small
.STACK
.code
TEST_WRITE_HEX PROC
        MOV DL,3Fh          ;Move o valor 3Fh para o registador DL
        CALL WRITE_HEX      ;Chama a sub-rotina
        MOV AH,4CH          ;Função 4Ch
        INT 21h             ;Retorna o controlo ao DOS
TEST_WRITE_HEX ENDP         ;Finaliza o procedimento
PUBLIC WRITE_HEX            ;Este procedimento converte para hexadecimal o byte
                            armazenado no registador DL e mostra o dígito Use:WRITE_
                            HEX_DIGIT
WRITE_HEX PROC
        PUSH CX             ;coloca na pilha o valor do registador CX
        PUSH DX             ;coloca na pilha o valor do registador DX
        MOV DH,DL           ;move o valor do registador DL para o registador DH
        MOV CX,4            ;move o valor 4 para o registador CX
        SHR DL,CL
CALL WRITE_HEX_DIGIT        ;mostra na tela o primeiro número hexadecimal
        MOV DL,DH           ;move o valor do registador DH para o registador DL
        AND DL,0Fh
CALL WRITE_HEX_DIGIT        ;mostra na tela o segundo número hexadecimal
        POP DX              ;retira da pilha o valor do registador DX
        POP CX              ;retira da pilha o valor do registador CX
        RET                 ;Retorna o controlo ao procedimento que chamou
WRITE_HEX ENDP
PUBLIC WRITE_HEX_DIGIT      ;Este procedimento converte os 4 bits mais baixos do
                            registador DL para um número hexadecimal e mostra-o no
                            ecrã do computador. Use: WRITE_CHAR
WRITE_HEX_DIGIT PROC
```


```
        PUSH DX             ;coloca na pilha o valor de DX
        CMP DL,10           ;compara se o número de bits é menor do que 10
        JAE HEX_LETTER      ;se não, salta para HEX_LETER
        ADD DL,"0"          ;se sim, converte para número
        JMP Short WRITE_DIGIT ;escreve o caracter
HEX_LETTER:
        ADD DL,"A"-10       ;converte um caracter para hexadecimal
WRITE_DIGIT:
        CALL WRITE_CHAR     ;imprime o caracter no ecrã
        POP DX              ;Retorna o valor inicial do registador DX
                            ;para o registador DL
        RET                 ;Retorna o controlo ao procedimento que chamou
WRITE_HEX_DIGIT ENDP
PUBLIC WRITE_CHAR           ;Este procedimento imprime um caracter na tela usando o D.O.S.
WRITE_CHAR PROC
        PUSH AX             ;Coloca na pilha o valor do registador AX
        MOV AH,2            ;Função 2h
        INT 21h             ;Interrupção 21h
        POP AX              ;Extrai da pilha o valor de AX
        RET                 ;Retorna o controlo ao procedimento que chamou
WRITE_CHAR ENDP
END TEST_WRITE_HEX          ;Finaliza o programa
```

4º Exemplo

```
;nome do programa: four.asm
.model small
.stack
.code
TEST_WRITE_DECIMAL PROC
        MOV DX,12345        ;Move o valor decimal 12345 para o registador DX
        CALL WRITE_DECIMAL  ;Chama o procedimento
        MOV AH,4CH          ;Função 4Ch
```

```
        INT 21h                 ;Interrupção 21h
TEST_WRITE_DECIMAL ENDP         ;Finaliza o procedimento
PUBLIC WRITE_DECIMAL            ;Este procedimento escreve um número de 16 bit
                                como um número sem sinal em notação decimal
                                ;Use: WRITE_HEX_DIGIT ;
WRITE_DECIMAL PROC
        PUSH AX                 ;Põe na pilha o valor do registador AX
        PUSH CX                 ;Põe na pilha o valor do registador CX
        PUSH DX                 ;Põe na pilha o valor do registador DX
        PUSH SI                 ;Põe na pilha o valor do registador SI
        MOV AX,DX               ;move o valor do registador DX para AX
        MOV SI,10               ;move o valor 10 para o registador SI
        XOR CX,CX               ;coloca a zero o registador CX
NON_ZERO:
        XOR DX,DX               ;coloca a zero o registador CX
        DIV SI                  ;divisão entre SI
        PUSH DX                 ;Põe na pilha o valor do registador DX
        INC CX                  ;incrementa CX
        OR AX,AX                ;não zero
        JNE NON_ZERO            ;salta para NON_ZERO
WRITE_DIGIT_LOOP:
        POP DX                  ;Retorna o valor em modo reverso
        CALL WRITE_HEX_DIGIT    ;Chama o procedimento
        LOOP WRITE_DIGIT_LOOP   ;loop
END_DECIMAL:
        POP SI                  ;retira da pilha o valor do registador SI
        POP DX                  ;retira da pilha o valor do registador DX
        POP CX                  ;retira da pilha o valor do registador CX
        POP AX                  ;retira da pilha o valor do registador AX
        RET                     ;Retorna o controlo ao procedimento que chamou
WRITE_DECIMAL ENDP              ;Finaliza o procedimento
PUBLIC WRITE_HEX_DIGIT          ;Este procedimento converte os 4 bits mais baixos
```


```
                                do registador DL num número hexadecimal e
                                imprime-os. Use: WRITE_CHAR ;
WRITE_HEX_DIGIT PROC
        PUSH DX                 ;Põe na pilha o valor do registador DX
        CMP DL,10               ;Compara o valor 10 com o valor do registador DL
        JAE HEX_LETTER          ;se não, salta para HEX_LETER
        ADD DL,"0"              ;se é, converte em dígito numérico
        JMP Short WRITE_DIGIT   ;escreve o caracter
HEX_LETTER:
        ADD DL,"A"-10           ;converte um caracter para um número
                                hexadecimal
WRITE_DIGIT:
        CALL WRITE_CHAR         ;mostra o caracter no ecrã
        POP DX                  ;Retorna o valor inicial para o registador DL
        RET                     ;Retorna o controle ao procedimento que chamou
WRITE_HEX_DIGIT ENDP
PUBLIC WRITE_CHAR               ;Este procedimento imprime um caracter no ecrã
                                usando uma função D.O.S. ;
WRITE_CHAR PROC
        PUSH AX                 ;Põe na pilha o valor do registador AX
        MOV AH,2h               ;Função 2h
        INT 21h                 ;Interrupção 21h
        POP AX                  ;Retira da pilha o valor inicial do registador AX
        RET                     ;Retorna o controlo ao procedimento que chamou
WRITE_CHAR ENDP
END TEST_WRITE_DECIMAL          ;finaliza o programa
```

Este programa mostra no ecrã os números 12345

5º Exemplo:

```
;nome do programa: five.asm
.model small
```

```
.stack
.code
PRINT_ASCII PROC
        MOV DL,00h              ;move o valor 00h para o registador DL
        MOV CX,255              ;move o valor decimal 255 para o registador CX
                                ;usado para fazer um laço com 255 interações
PRINT_LOOP:
        CALL WRITE_CHAR         ;Chama o procedimento que imprime
        INC DL                  ;Incrementa o valor do registador DL
        LOOP PRINT_LOOP         ;Loop para imprimir 10 caracteres
        MOV AH,4Ch              ;Função 4Ch
        INT 21h                 ;Interrupção 21h
PRINT_ASCII ENDP                ;Finaliza o procedimento
WRITE_CHAR PROC
        MOV AH,2h               ;Função 2h para imprimir um caracter
        INT 21h                 ;Imprime o caracter que está em DL
        RET                     ;Retorna o controlo ao procedimento que chamou
WRITE_CHAR ENDP                 ;Finaliza o procedimento
END PRINT_ASCII                 ;Finaliza o programa
```

Este programa mostra no ecrã o valor dos 256 caracteres do código ASCII.

Bibliografia

GOUVEIA, José e MAGALHÃES, Alberto, *Curso Técnico de Hardware*. Lisboa: FCA, 2002.

GOUVEIA, José e MAGALHÃES, Alberto, *Hardware: Montagem, Actualização, Detecção de Avarias em PC's e Periféricos*. Lisboa: FCA, sd.

GOUVEIA, José e MAGALHÃES, Alberto, *Hardware para PC's e Redes.* Lisboa: FCA, sd.

University of Guadalajara, tutorial da linguagem assembly Information Sistems General Coordination, Culture and Entertainment Web Copyright(C) 1995-1996.