Apostila de

Linguagem

de Montagem

Capítulos:

Índice:

1. Introdução ao 8088/86

1.1. Historia da evolução da família 80-XXX

Desenvolvido pela INTEL no ano de 1972 o primeiro microprocessador , possuía o código 4004, a partir de então o desenvolvimento dos microprocessadores não parou mais, a tabela abaixo mostra esta evolução

Fabricante	Microprocessador	Comprimento da palavra (bits)	Capacidade de endereçamento
INTEL	4004	4	8K
MOTOROLA	6800	8	64K
INTEL	8080	8	64K
INTEL	8085	8	64K
ZILOG	Z80	8	64K
INTEL	8088/86	8/16	1M
MOTOROLA	68000	16	16M
INTEL	80286	16	
INTEL	80386	16	
MOTOROLA	68020	32	
INTEL	80486	32	
INTEL	PENTIUM	32	

Figura 1: A evolução dos microprocessadores.

O 8088/86 teve como pai o processador 8085, e para não serem perdidos os programas feitos para o 8085 , algumas soluções foram feitas no 8088 para que este mantivesse a sua compatibilidade, com isso acarretou em problemas de segmentação da memória, tamanho máximo de segmento de dados e código. Na mesma época a MOTOROLA , fabricante do processador 6805, o qual equipava os apples II, produziu um processador novo de 16 bits reais (barramento externo e interno), não mantendo a compatibilidade com o seu antecessor, este processador passou a equipar o **MAC** fabricados pela "Apple".

1.2 Estrutura interna da CPU

figura 2 : Estrutura de blocos da CPU.

A CPU pode ser dividida em três grandes blocos: Registradores, Unidade de Controle e Unidade de lógica e Aritmética (ULA).

Unidade de Controle: Executa as seqüências de instruções de um programa na ordem correta. Controla o fluxo de dados, busca de novas instruções e dados.
A função da UC é receber a instrução a executar, trazida da memória, interpretá-la (decodificação) e então gerar os sinais de controle que comandem a operação de hardware.
Além de executar a instrução corrente estes sinais devem comandar a busca da próxima instrução a executar, o que pode ser um processo complexo.
A maioria das instruções de um programa são executadas seqüencialmente, com a próxima instrução estando localizada imediatamente após a instrução em execução. O mecanismo normal de passagem de uma instrução para a seguinte é através do incremento do registrador PC (ou seu equivalente) ao fim da instrução corrente.
PC <= endereço da próxima instrução.
Entretanto, muitas vezes é necessário que a próxima instrução a executar não seja a imediatamente subsequente à instrução atual. Para que isto ocorra devem existir maneiras de se carregar o PC com o endereço não subsequente, possibilitando que no programa ocorram "saltos" durante a execução. Estes são chamados desvios (condicionais e incondicionais).
É também útil que se possa retornar ao ponto original da execução após ter efetuado um "salto". Isto é obtido através de instruções auxiliares que permitem o armazenamento do PC antes do "salto" e sua recuperação posterior. Este é o mecanismo das subrotinas.
Por fim, é também necessário que eventos externos ao programa, como por exemplo comandos de teclado, possam interromper o programa em execução para seu atendimento, após o que o fluxo normal é restabelecido. Este é o mecanismo chamado de interrupção.

Unidade de Lógica e Aritmética : A ULA de um processador realiza o conjunto de operações lógicas e aritméticas definidas pelas instruções de manipulação de dados. Todas as instruções podem ser executadas através do conjunto mínimo de operações descrito a seguir.

- **Adição Binaria :** A adição é a operação aritmética básica dos computadores, através da qual as demais operações podem se executadas.

- **Operações Booleanas :** São as operações lógicas, and, or, not e xor, presentes também em linguagem de alto nível.

- **Complemento :** Forma de representação de números binários negativos. A principal vantagem da representação em complemento é que ela permite qual as subtrações possam ser feitas através de adições com o complemento.

- **Deslocamento de bits :** O deslocamento, à esquerda ou à direita, dos bits em uma palavra permite que os dados de qualquer tipo sejam manipulados bit-a-bit.

Como resultado das operações executadas a ULA altera bits do registrado de estados (FLAG) do processador. É através desta alteração que uma instrução pode saber qual o resultado da operação executada pela instrução anterior, permitindo assim o programa

altere o seu fluxo de execução de acordo com os dados sobre os quais está operando. Em geral, o processador possuem, no mínimo, os seguintes flags:

Carry. É o bit de "vai-um". Quando uma soma um resultado gera um bit a mais do que o espaço reservado, este bit é armazenado no bit de carry, podendo ser usado posteriormente para obtenção do resultado final. O carry também é usado nas subtrações, quando passa a ter o significado de borrow (falta-um).

Zero. Bit que contem 1 quando o resultado final de uma instrução for zero.

Signal. Reproduz o bit mais significativo do resultado. Como visto a seguir, através deste bit pode-se determinar o sinal de um número.

Overflow. Bit usado para indicar o estouro de uma operação. E ativado quando o penúltimo bit de um resultado transfere um carry para o ultimo bit. Este tipo de definição faz sentido quando usa-se o último bit apenas como indicador de sinal.

Registradores : Um conjunto de posições de memória chamados registradores, para o armazenamento temporário de dados e instruções quando estes são usados pelo processador.

1.3. Detalhes dos registradores

Pode ser muito devagar para o processador trabalhar somente com a memória primária. Cada Byte que é utilizado precisa ser transferido de e para a memória. Entretanto, há uma pequena quantidade de memória construída dentro do processador.

Esta memória consiste de 14 palavras de memória de escrita e leitura. (palavra = 2 bytes) Cada uma dessas palavras é chamada de registrador. Cada registrador tem o seu nome próprio e seu próprio propósito.

Na tabela abaixo, temos uma lista dos registradores e suas abreviações. Quando você escreve um programa e usa depuradores (debuggers), você faz referencia aos registradores pela sua abreviação.(O flag não possui abreviação). Também temos as divisões de suas categorias que podem ser : Uso Geral, Offset e Segmento.

Nome dos Registradores, Abreviações e Categorias.		
Nome	Abreviação	Categoria
Acumulador	AX(AH,AL)	Uso geral
Registrador Base	BX(BH,BL)	Uso geral
Registrador Contador	CX(CH,CL)	Uso geral
Registrador de Dado	DX(DH,DL)	Uso geral
Ponteiro da Base	BP	Offset
Ponteiro de instrução	IP	Offset
Ponteiro do Stack	SP	Offset
Índice de Destino	DI	Offset
Índice de Fonte	SI	Offset
Segmento de Dado	DS	Segmento
Segmento Extra	ES	Segmento

Segmento do Stack	SS	Segmento
Segmento de Código	CS	Segmento
Flags	(não possui)	(não possui)

Figura 3: Registradores

Há quatro registradores de uso geral: AX, BX, CX e DX. Estes registradores oferecem um armazenamento temporário conveniente para qualquer tipo de informação. Ele são especialmente muito úteis em operações aritméticas. Por serem construídos dentro do processador, as intrusões que os utilizam são mais rápidas que as que utilizam a memória primaria.
Você pode utilizar qualquer um desses registradores para agir como "pau para toda obra". Entretanto , AX, BX e CX também possuem funções especificas.

AX é o principal registrador utilizados nas funções aritméticas (embora todos os registradores de uso geral possam ser utilizados). AX é freqüentemente utilizado para acumular o resultado de um calculo. Por esta razão é chamado de **acumulador**.

BX (em conjunto com BP) é as vezes chamado de registrador base porque ele pode ser utilizado para manter endereços bases (explicado nos próximos capítulos)

CX é usado com certas instruções para realizar operações repetidas. Em certos casos, CX deve conter o número de vezes que você deseja repetir a operação. Devido a isto ele é referenciado como **Registrador Contador**.

DX é algumas vezes chamado de **registrador de dado** por que é usado para manter dados de qualquer natureza.

Os registradores de uso geral podem ser divididos em 2 partes e acessadas separadamente. As letras "H" e "L" servem para identificar High e Low. Exemplo: O registrador AX pode ser dividido em 2 registrador de 8 bits - AH e AL. podendo ser realizadas operações com 1 byte sobre eles, quando ha necessidade do uso de uma palavra utilizamos o registrador na sua forma inteira (AX). Exemplo da figura 4 nos ilustra o relacionamento entre os registradores AX -> AH,AL

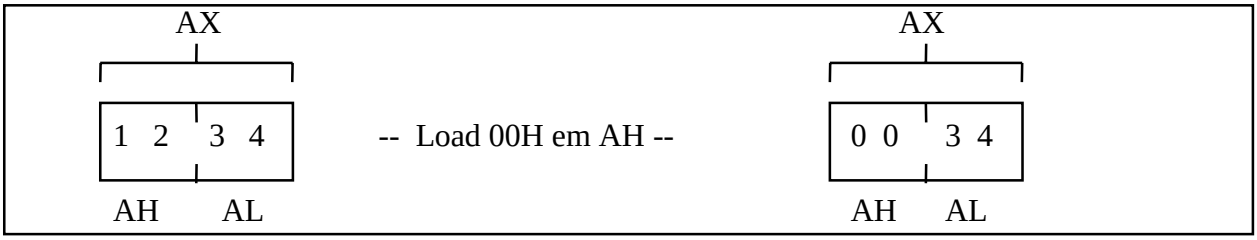


Figura 4 - Relacionamento entre AX e AH,AL

1.4. Noções sobre Temporização

A operação interna de um computador durante a execução das instruções é sincronizada, permitindo que as partes do hardware atuem de forma independente desde que sejam respeitados os prazos de cada tarefa.

Como os sinais elétricos em um computador só assumem dois valores lógicos: 0 e 1, é possível descrever a relação entre estes sinais, através de gráficos de Amplitude x Tempo, úteis para refletir a relação de causa e efeito entre os sinais.

Ciclo de Maquina :
O computador opera sincronizado por um sinal periódico chamado de clock (relógio), uma onda quadrada que altera níveis lógicos 1 e 0. Um período do clock é chamada de Ciclo de Maquina. (figura 5)

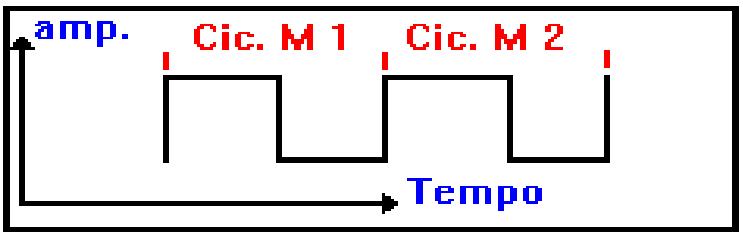


Figura 5. Ciclo de Maquina

Exercícios:

1) **Qual é a estrutura interna da CPU.**
2) **Descreva sucintamente as unidades da estrutura interna da CPU.**
3) **Cite três registradores de uso geral e três de uso especifico.**
4) **Os registradores de uso geral (16 bits) podem ser divididos em quantos registradores de 8 bits e qual a sua nomenclatura.**
5) **O que você entende por ciclo de máquina.**

2. O que vem a ser Linguagem de Montagem

A linguagem de montagem (em inglês *assembly language*) permite representar as seqüências de números binários contendo as instruções entendidas pelo processador (linguagem de maquina) através de símbolos representativos das instruções, operadores e endereços. No caso do 8086, as instruções correspondem a representação simbólica de comandos armazenados diretamente na pastilha microprocessadora, quando de sua fabricação.

A geração do código em linguagem de máquina é feita usando-se programas tradutores, os montadores (“assemblers”) e ligadores (“linkers”).

Alem das instruções que geram código de maquina para o processador, os montadores aceitam linhas de instrução no programa-fonte que contenham comandos para ordenar o processo de montagem, os quais fornecem informações que o montador não poderia deduzir por si só. Por não gerarem código de maquina, estas linhas são chamadas de pseudo-instruções ou *diretivas*

2.1. Formato de uma instrução

Cada instrução ou diretiva ocupa uma linha do programa-fonte, dividida em quatro colunas chamadas campos:

ROTULO	MNEMÔNICO	ARGUMENTO	COMENTÁRIO

Campo de Rotulo. É de conteúdo opcional, representando simbolicamente o endereço da linha de instrução. O conteúdo deste campo é tratado como um nome a ser usado por outras instruções que se referenciem a esta linha de programa, como por exemplo, nomes de constantes ou variáveis as quais são atribuídos.

ROTULO	MNEMÔNICO	ARGUMENTO	COMENTÁRIO
inic	mov	ax,[bx]	

Campo de Mnemônico. É o único campo de conteúdo obrigatório em uma linha. Nele esta um conjunto de letras representando a operação que deve ser feita pela instrução. Em linhas de diretivas este campo contem o mnemônico da diretiva.

ROTULO	MNEMÔNICO	ARGUMENTO	COMENTÁRIO
	.MODEL	small	; define tam. dos segmentos
	push	cx	

Campo de Argumento. Contem os operandos sobre os quais atua a operação definida no campo de mnemônico. São aceitos como operadores registradores da CPU, endereços de memória e E/S, expressões e constantes.

ROTULO	MNEMÔNICO	ARGUMENTO	COMENTÁRIO
		AI, [bx + 4]	

add

Campo de Comentário. O conteúdo deste campo é ignorado pelo montador. Sua utilidade é como área para informações que tornem o programa mais fácil de ser compreendido. Na sintaxe dos exemplos o comentário é sempre iniciado por **";"**.

ROTULO	MNEMÔNICO	ARGUMENTO	COMENTÁRIO
	.MODEL	small	; define tam. dos segmentos

Diretivas para definição de variáveis e constantes.

O grande número de posições de memória requer um mecanismo de nomear individualmente as partes em uso por um programa o que é feito por comandos chamados diretivas.(ver apêndice)

Variáveis:

A criação de variáveis é feita através das diretivas de variáveis. Esta são como as declarações de variáveis das linguagens de alto nível, reservando espaço em memória para a variável definida.

identificador	diretiva de variável	espaço de memória

Constantes:

Muitas vezes, não se deseja apenas reservar espaço em memória mas também armazenar um valor fixo a ser usado freqüentemente. Isto é feito usando-se uma diretiva de constante, que faz uma atribuição de um dado valor (o qual pode ser literal), a uma posição de memória alocada para a constante.

identificador	diretiva da constante	valor

2.2. Arquitetura do conjunto de instruções

A porção mais interna do computador visível para o programador é o seu conjunto de instruções em linguagem "assembly". O numero máximo de endereços permitidos por instrução esta relacionado com os recursos do "hardware" e determina o tipo de arquitetura do conjunto de instruções.
Seja o caso em que se deseja a soma do conteúdo de duas variáveis , X e Y, com o resultado sendo armazenado em Z.

```
Z :=  X + Y;
```

Se X, Y e Z representam posições de memória, a linha de programa pode ser representada por uma única instrução em linguagem “assembly” que contenha os endereços das três variáveis.

```
ADD   Z,X,Y
```

Instruções com endereçamento completo, operandos e destino, exigem que se tenha um grande aumento de bits por instrução reservados para endereço aumentando o custo da memória. Alem disso, muitas vezes não é vantajoso que o resultado retorne para a memória como no caso de resultados intermediários que devem alimentar as instruções seguintes, podendo pois, permanecer no processador para evitar buscas desnecessárias de dados na memória. Fato similar ocorre com os operandos que nem sempre precisam ser preservados na memória após a execução de uma instrução. As opções à arquitetura de três endereços são :

Arquitetura Baseada em Registradores de Propósito Geral. Somente operandos são mencionados explicitamente na instrução, sejam eles registradores internos do processador ou posições de memória. O resultado é armazenado no endereço de um dos operandos, destruindo o conteúdo anterior. Este tipo de maquina é conhecida com arquitetura de dois endereços.

Arquitetura Baseada em Acumulador. Usa-se para as operações, um registrador especial no processador, chamado acumulador, que deve conter um dos operandos da instrução e ao final desta o seu resultado. O conjunto de instruções deve prover duas operações, LOAD e STORE para escrever e ler respectivamente o acumulador. Esta é a arquitetura de um endereço pois apenas o outro operando que não o acumulador precisa ser mencionado na instrução.

Arquitetura Baseada em Pilha. Define-se uma região da memória , chamada pilha, para o armazenamento temporário dos operandos e resultados. A pilha é uma estrutura do tipo LIFO, onde o ultimo elemento escrito (topo da pilha) , é o primeiro a ser lido. É necessário um registrador interno ao processador, o SP - “stack pointer”, para apontar o endereço do topo da pilha, e duas instruções, PUSH e POP para a escrita e leitura no topo da pilha. Sempre que usadas estas instruções atualizam o SP. Este tipo de maquina é chamado arquitetura de zero instruções porque não referencia aos operandos ou resultados na instrução.

A arquitetura baseada em registradores de propósito geral é a mais usada devido a flexibilidade que oferece ao compilador para armazenamento temporário de dados no processador. A arquitetura de acumulador só possuindo um registrador interno, requer um trafego de dados maior entre processador e memória. A arquitetura de pilha possui a mesma limitação quanto ao trafego, agravada pelo fato de ser impossível o acesso randomico à pilha.

2.3 Modos de endereçamento.

As maneiras pelas quais uma instrução pode designar:

- Endereço de memória, registrador e E/S.

- Endereços de desvio

Os principais modos de endereçamento são descritos a seguir:

Endereçamento imediato. O operando é parte da instrução

```
Add     ax, 10h
```

Endereçamento Direto. O endereço do operando é parte da instrução

```
Add     ax,[8000h]
```

Endereçamento Indireto (implícito). A instrução contem o local onde esta o endereço do operando

```
Add     ax, [bx]
```

Endereçamento Indexado. O endereço do operando é gerado adicionando-se um valor de índice ao endereço contido na instrução. Este índice em geral esta armazenado em um registrador do processador especifico para esta função.

```
Add     ax, [ bx + di]
Add     ax, [bp + 4]
```

2.4 Como se faz comentários

Para se inserir um comentário em um programa assembly utilizamos **";"**.
Uma boa pratica de programação é usar comentários para documentar programas em "assembly", procurando tornar mais claros pontos importantes do algoritmo implementado. Muitos fabricantes de microprocessadores sugerem que a cada rotina escrita seja acrescentado um cabeçalho, formado por linhas contendo apenas o campo de comentário, onde a rotina é sumarizada.

```
;*****************************************************************
; Nome :  INICIA_BUF
; Descrição:  Carrega dado de AL em buffer na memória
;             CX contém número de posições do buffer
;             DI contém endereço inicial do buffer
;
; Entrada : AL,CX, DI.
; Saída:    buffer[DI] ate buffer[DI+CX]
;
; Destroi:   CX, DI, buffer
;*****************************************************************
```

```
        .MODEL    small          ; organiza segmentos
inic    mov       [di],al        ; armazena dado em buffer[di]
        inc       di             ; incrementa apontador
        dec       cx             ; posições restantes no buffer
        jnz       inic           ; repete se cx<> 0
        end                      ; fim do programa.
```

O comentário não deve ser uma tradução e sim sumarizar a sua intenção (Objetivo do trecho).

2.5 Ferramentas de Programação

- **Tradutores :** Programas que transformam um programa de entrada (programa-fonte), escrito em uma linguagem de programação, para seu equivalente em outro formato (programa-objeto).

- **Compiladores :** Tradutores onde o programa-fonte está escrito em uma linguagem de alto nível e o programa-objeto é gerado em linguagem de máquina ou de montagem (“assembly”).

- **Pré-Processadores :** Tradutores que convertem um programa de uma linguagem de alto nível para outra.

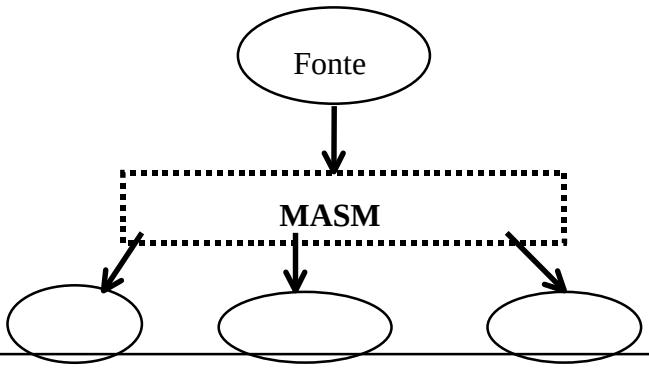
- **Interpretadores :** Programas que aceitam um programa-fonte em uma dada linguagem de alto nível e o executam, em geral obedecendo a seqüência:

  1. Conversão do fonte para um formato interno.
  2. Execução do programa gerado no formato interno.
  3. Destruição do programa gerado no formato interno.

- **Monitores :** Conversores de um programa em linguagem de maquina escrito em octal ou hexadecimal, para binário.

- **Montadores :** Tradutores de programas em linguagem de montagem para linguagem de máquina.

No nosso caso o MASM (“Microsoft Macro Assembler”), montador para o IBM-PC, gera três arquivos de sadia, a partir do arquivo-fonte (extensão ASM):



Objeto Referência Listagem

Figura 6: MASM

Arquivo-Objeto(.Obj). Arquivo de saída para o ligador (Link). Contem a tradução do arquivo-fonte em linguagem de máquina. Não é legível com um editor por não ser um arquivo texto.

Arquivo de Referencias Cruzadas(.CRF). Contem informações sobre nomes usados no programa. Onde estes foram definidos e em que linha de programa aparecem como argumentos. É útil para a localização de variáveis e rótulos em um programa.

Arquivos de Listagem(.LIS). É um arquivo-texto numerado linha a linha, contendo para cada linha, a instrução em linguagem de montagem e seu correspondente em linguagem de máquina. As linhas indicadas nas mensagens de erro do MASM se referem as linhas do arquivo de listagem.

2.6 Montador

O processo de tradução de programas escritos em língua de montagem para a linguagem de maquina é chamado *montagem* e o programa que executa a tradução , *montador*.
A montagem é um dos passos da cadeia vista na figura 7, a qual produz um programa executável. O processo é iniciado pelo programador que escreve um programa-fonte em linguagem de montagem, usando um editor de texto; este programa serve como entrada para o montador que gera o programa-objeto, o qual é um programa com instruções em linguagem de maquina porem ainda não executáveis; por fim , usa-se o ligador, programa que recebe um ou mais programas-objetos como entrada e gera o programa executável.

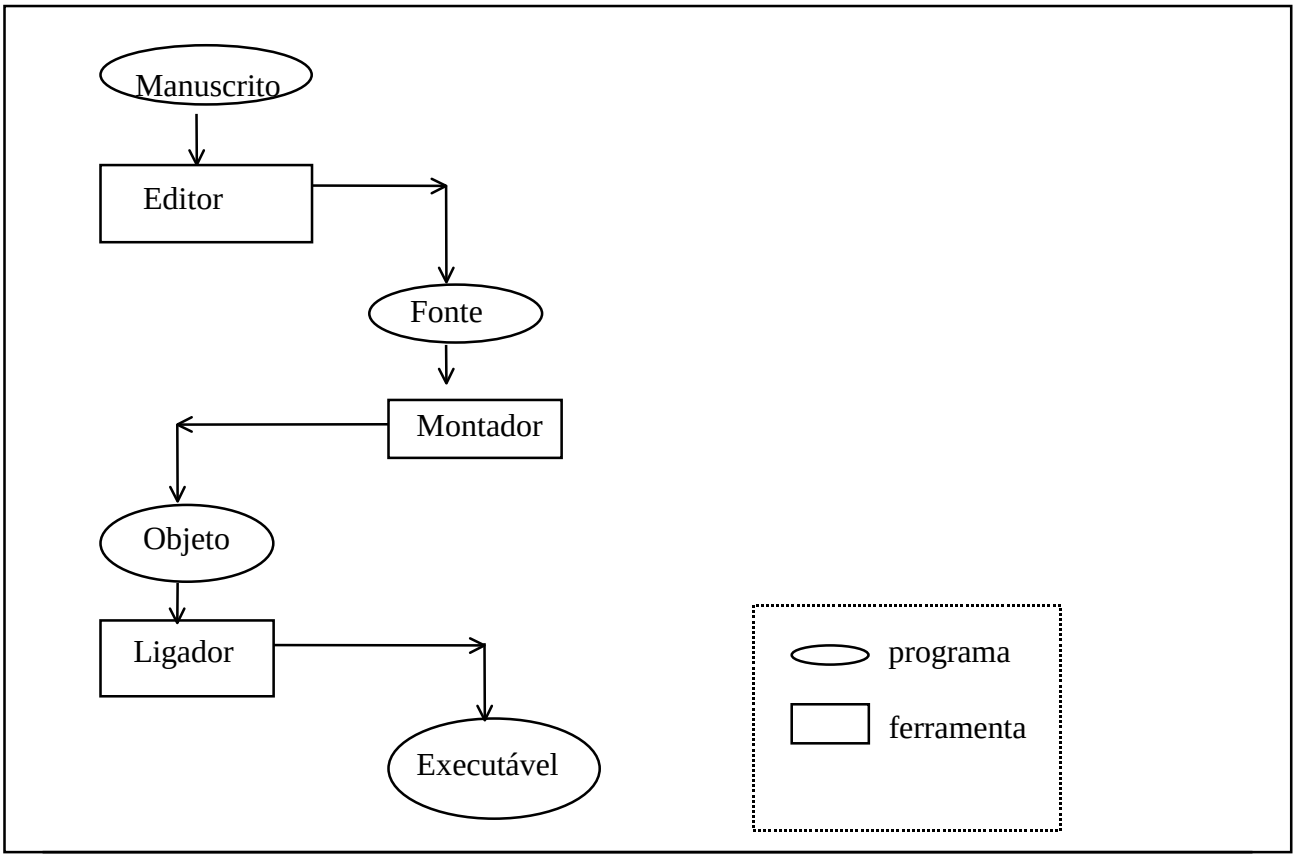


Figura 7 : Passo para a geração de um programa em linguagem de montagem.

Exercício : Montagem de um programa com o MASM

O programa a seguir demostra os passos para o uso do MASM , ilustrando ainda alguns conceitos importantes como o uso de *pseudo-instruções* e *chamadas de serviços do DOS.*

1. Edição

Usando um editor que produza um arquivo não formatado, edite o programa:

```
        .MODEL      small
        .STACK      100h
        .DATA
mensagem    DB          'Alo Mundo', 13, 10, '$'
        .CODE
        mov         ax,@data
        mov         ds,ax
        mov         ah,9
        mov         dx, OFFSET mensagem
        int         21h
        mov         ah,4Ch
        int         21h
        END
```

terminado a edição salve o arquivo com o nome que quiser (no caso) hello.asm

2. Montagem

Chame o MASM usando o comando : MASM hello

será gerado o programa hello.obj, um arquivo binário intermediário.

3. Ligação

Gera uma versão executável do programa, evocando o ligador:

link hello

A partir da versão 6 do MASM , podemos utilizar um programa só para montar e linker, o qual vem junto com o MASM. Ele se chama ML (Monta e linka).
Para utilizarmos, devemos chama-lo da seguinte maneira :

ml hello.asm

parâmetros para o ML :
/Fl - cria o arquivo de listagem.
/Fm - cria o arquivo de map.
/Fr - cria a referência cruzada.

4. Execução

É só chamar pelo nome do arquivo. hello

2.7 Diretivas. (Constantes, Variáveis e Expressões)

- **Definições de Constantes**

Constantes são números, cadeia de caracteres ASCII, expressões com valor definido, etc. A atribuição de um valor, a um nome é possível usando-se o símbolo "=" ou a diretiva "EQU" (equal).

```
base  =    bx + 3
valor =    4
valor EQU  valor * 3
```

Como efeito das atribuições, em todas aparições dos nomes **base**, valor e valor2 no programa, o montador os substituirá pelas constantes correspondentes. Isto vale para a ultima linha do exemplo, onde **valor2** é calculado a partir de **valor**. Uma atribuição de constante pode ser alterada a qualquer momento em um programa.

- **Alocução de Memória para Variáveis**

As variáveis de um programa ocupam área física na memória, a qual pode ser definida através das diretivas DB, DW, DD, DF, DQ e DT. A tabela abaixo mostra o numero de bytes alocados por varias diretivas

DIRETIVA	DB	DW	DD	DQ	DT
BYTES	1	2	4	8	10

De acordo com a diretiva, diferentes quantidades de bytes são alocadas:

DB pode ser usado para :

- Uma expressão com valor entre -127 e + 255.

```
teste    DB    10h
```

- Uma cadeia com um ou mais caracteres. Neste caso um byte é definido para cada caracter da cadeia.

```
cadeia   DB    "Cadeia de Caracteres"
```

No caso do 8086, DW, DD, DQ, DF e DT podem armazenar constantes numéricas variando entre -32768 e + 65535 e endereços com até 16 bits. Para cadeias de caracteres há diferenças:

DW: cadeias de até 2 bytes.
DQ: cadeias de até 4 bytes.
DD: cadeias de até 8 bytes.
DT: cadeias de até 10 bytes.

É possível definir-se variáveis não iniciadas, através do símbolo **?**.

```
indefinido      DB      ?
```

se quisermos repetir uma declaração de tamanho podemos usar o operador DUP.

```
repetição       DB      2 DUP(7,3)
rep             DB      100 DUP (1,2,?)
```

- **Expressões Aritméticas**

As quatro operações aritméticas básicas são permitidas em expressões, sendo aceitos tanto variáveis como números nas operações. O resultado é sempre associado a uma variável.

```
difprod         EQU     5*(4 - 6)
novores         =       difprod / 3
```

2.8 Gerência de Memória

Em todo acesso à memória o 8086 usa ao menos um dos registradores de segmento: CS, DS, SS ou ES. Como estes registradores permitem o acesso direto a, no máximo, 64k endereços por registrador, caberia ao programador definir quais serão usadas por dados e programas. Para facilitar esta tarefa, os montadores possuem diretivas que permitem definir e manipular os segmentos sem grande ônus ao esforço de programação.

MODEL - Segmentação Simplificada

Os montadores MASM e TASM possuem modelos de segmentação como vistos na tabela 8, que são úteis na maioria dos programas. Para todos os modelos, o espaço reservado para a área de pilha é de 200h palavras (1k byte). Para fazer uso de um dos modelos basta iniciar o programa-fonte com a diretiva .MODEL, seguida do modelo adotado.

	MODELO			
	small	medium	compact	large
DADOS	64 kB	> 64 kB	64 kB	> 64 kB
PROGRAMAS	64 kB	64 kB	> 64 kB	> 64 kB

figura 8 : Modelos comuns de segmentação.

Uma vez definidos os tamanhos para os dados e programas, é preciso dividir o programa-fonte, separando estas áreas por diretivas que indiquem ao montador como usar os segmentos:

.**DATA** : Inicia ou continua a parte do programa-fonte contendo dados com valor inicial.

.**CONST** : Inicia ou continua a parte do programa-fonte contendo dados constantes. É similar a .data.

.**DATA**? : Inicia ou continua a parte do programa-fonte contendo dados sem valor inicial.

.**STACK** <tam> : Começa ou continua o segmento de pilha. O parâmetro <tam>, é opcional e define o número de palavras reservadas para a pilha. Se não especificado é adotado o valor 200h.

.**FARDATA** <nome> : Inicia ou continua a parte do programa-fonte contendo dados com valor inicial que não pertencem ao segmento de dados Básico (chamado *near*). Isto é necessário se a área de dados ocupa mais do que um segmento físico (> 64 kB). Os dados armazenados em segmentos adicionais (chamados *far*) são definidos com o uso de ".FARDATA <nome>", onde *nome* é o nome do segmento. Um programa pode ter mais do que um segmento do tipo *far* para dados.

.**FARDATA**? <nome> : O mesmo que .fardata só que para dados sem valor inicial.

.**CODE** <nome> : Inicia ou continua a área de código do programa. Nos casos em que a área de programa ocupa mais do que um segmento físico, os segmentos adicionais (far), são definidos com o uso de ".CODE <nome> ", onde nome é o nome do segmento. Um programa pode ter mais do que um segmento do tipo far para instruções.

Quando usamos a diretiva .model, o montador armazena o endereço inicial dos segmentos nas variáveis especiais, listadas abaixo:

VARIÁVEL	SIGNIFICADO
@code	segmento reservado para CS
@data	segmento reservado para DS
@stack	segmento reservado para a pilha (SS)
@fardata	segmento do dado tipo "far" corrente
@fardata?	segmento de dado "far" corrente não iniciado
@curseg	segmento corrente

Tabela 9: Variáveis para endereçamento dos segmentos

Um exemplo da utilização das variáveis acima citadas pode ser visto no código abaixo. Em um programa, dados armazenados no segmento tipo far (assumindo que houvesse apenas um), podem ser acessados desde que o registrador DS seja previamente carregado com o endereço inicial do segmento.

```
mov    ax,@fardata          ; lê a variável @fardata
mov    ds,ax                ; muda DS para @fardata
```

Quando a diretiva .**model** é usada pode-se iniciar os registradores DS, SS e SP através da diretiva **.startup**. Após esta diretiva, DS e SS ocuparão endereços dentro da área de memória física definida por @data, enquanto que SP ser carregado com o endereço final da pilha.

SEGMENT - Segmentação Genérica.

Em alguns casos o programador pode desejar definir os segmentos de modo diferente daqueles disponíveis nos modelos simplificados. Para essas situações deve-se usar a diretiva SEGMENT, com a seguinte sintaxe:

```
<nome>        SEGMENT    [atributos]
```

Cada segmento é identificado por um nome que precede a palavra reservada segment. Os atributos são opcionais e dizem respeito basicamente a como o ligador deve unir os segmentos e como eles devem ser dispostos na memória. Atributos como CODE, DATA, DATA? ou STACK indicam a natureza das informações do segmento. Já os atributos PUBLIC e PRIVATE habilitam ou inibem que segmentos com mesmo nome, em módulos diferentes sejam concatenados.
Um segmento sempre termina com uma diretiva de fim na forma:

```
<nome>        ENDS
```

Exercício :

1) Qual o formato de uma instrução? Explique cada campo ?
2) Quais os modos de endereçamento?
3) Por que há necessidade de se comentar um código Assembler e como se faz isso?
4) Cite as diretivas e de um exemplo :
   - Constantes
   - Alocar memória
5) O que significa .DATA?

Capitulo 3 As Instruções do 8088/86

Neste capitulo são descritas as instruções básicas do conjunto disponível para a família de microprocessadores INTEL x86.
O conjunto de instruções em linguagem de montagem pode ser dividido genericamente em três classes:

a) Instruções Lógico-Aritméticas : Formam o núcleo do conjunto de instruções tendo em vista que os computadores são usados primariamente para a solução de problemas matemáticos

b) Instruções de Transferência de Dados : A existência de módulos dedicados a execução das instruções (CPU), armazenamento de dados e programas (Memória) e interface com o exterior (E/S), requer instruções que permitam o movimento de dados e integrem os módulos.

c) Instruções de Controle de Fluxo : São usadas para a tomada de decisões durante execução do programa, permitindo desvios na seqüência de execução.

3.1 Instruções de Transferencia de Dados

Sob à óptica das transferencias de dados, o computador é composto pelos registradores internos da CPU e E/S; e pela memória, esta dividida em três partes conforme visto na figura 10.

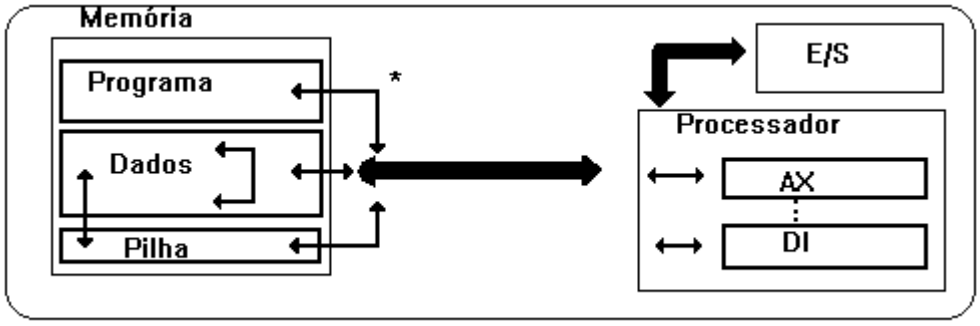


Figura 10: Transferência de Dados.

As setas indicam o fluxo possível através das instruções de transferencias. O caminho indicado por um * não se refere ao movimento de dados, mas sim a carga inicial do programa e a busca de instruções quando da sua execução.

3.1.1 Transferencia CPU <> CPU e CPU <> Memória de Dados

Mov destino, origem

Transferencias de dados de 8, 16 bits na formas: Registrador <> Registrador e Registrador <> Memória, conforme exemplificado abaixo.

Destino	ORIGEM			
	DADO	MEM	REG	REG_SEG
MEM	mov [cx],3168h	...	mov [si+4],ax	mov [2000h],ds
REG	mov cx, 3168h	mov al,[si]	mov ch,034h	mov ax,es
REG_SEG	...	mov cs,[bx+2]	mov ss,dx	...

Figura 11: transferência de blocos na memória

A rotina a seguir move o conteúdo de um buffer (bloco) na memória, cujo endereço inicial esta em SI, para o destino DI. O numero de elementos do buffer de origem está em CX.

```
buff:   mov     al,[si]     ;leitura do buffer de origem
        mov     [di],al     ; escrita no buffer destino
        inc     si          ; atualização dos ponteiros
        inc     di
        dec     cx          ; atualiza o contador
        jnz     buff        ; desvia se não chegou ao fim (cx=0)
        ret
```

XCHG operando, operando

Permite a troca de conteúdo entre operandos nas formas: Registrador <> Registrador e Registrador <> Memória, conforme tabela abaixo. No caso de um dos operandos ser o acumulador, este pode ser deixado implícito na instrução.

Operandos		
REG<>REG	ACUM<>REG	MEM<>REG
xchg bx,cx	xchg bx	xchg si,[dx+4]

Figura 12: Memória <> memória via XCHG.

As instruções a seguir permutam um dado da posição mem1, da memória , com outro da posição mem2

```
...
mov     al,mem1     ;lê mem1
xchg    al,mem2     ;transfere mem1 p/ mem2
mov     mem1,al     ;transfere mem2 p/ mem1
```

XLAT

Leitura de tabelas na memória, com ate 256 posições de 8 bits, usando os registradores AL e BX para calculo do endereço de memória para o acesso à tabela. O registrador de segmento usado é o DS e o valor lido é armazenado em AL, como visto abaixo.

XLAT	[AL] <- [[AL] + [BX]]

Exemplo : Conversão de tabelas usando XLAT

A rotina a seguir converte os dados existentes em um buffer na memória segundo os valores lidos nas posições correspondentes de uma tabela de conversão. BX contem o endereço da tabela de conversão, SI o endereço inicial do buffer de origem e DI o endereço equivalente do buffer de destino. CX contem o numero de elementos a ser convertido.

```
conv:   mov     al,[si]     ;carga do buffer de origem
        xlat                ;traducao
        mov     [di],al     ;dado convertido no buffer destino
        inc     si
        inc     di          ; incremento dos ponteiros
        dec     cx
        jnz     conv        ; fim da conversão ?
        ret
```

LEA/LDS/LES/LFS/LGS/LSS reg, mem

Carga de reg e reg_seg, conforme exemplos da tabela abaixo. São transferidos 32 bits da memória 16 para reg e os 16 seguintes para o registrador de segmento envolvido.

LEA bx, [si+F302h]	[bx] <- [[si] + F302h + [ds]]
LDS dx, [10h]	carga de DX e DS ,a partir de 10h
LES cx, [bx]	carga de CX e ES, a partir de [[BX] + [DS]]

LEA é muito utilizado para carregar o endereço efetivo da posição. Usando para apontar o primeiro endereço de um array.

3.1.2 Transferencia envolvendo a PILHA.

Transferencia de dados de 16 de registradores e da memória de dados , com a pilha. Os processadores da linha 80x86 usam a pilha da seguinte forma: o apontador de pilha (SP) endereça a pilha a partir da sua base (endereço final da área de pilha), sendo decrementado a cada escrita e incrementado a cada leitura. A memória é tratada como estando organizada em posições de um byte. A cada acesso à pilha ocorre uma das ações a seguir , de acordo com o caso.

push, pushf -> pilha -> pop, popf

Leitura de Registradores de 16 bits da Pilha. Os 16 bits armazenados no topo da pilha, no endereço apontado por SP (e SP -1), são transferidos para o registrador da CPU indicando na instrução. O SP é então incrementado em 2 posições, passando a apontar o próximo dado da pilha. O procedimento se repete no caso da instrução fazer referência a mais do que um registrador.
Escrita da Registrador de 16 bits na Pilha. O apontador de pilha SP é decrementado em duas posições, para apontar o próximo endereço vago na pilha, em seguida o(s) registrador(es) referenciado(s) é(são) escrito(s) na pilha. ficando SP com o endereço do ultimo registrador escrito.

A escrita na pilha é feita usando as instruções **PUSH**, **PUSHF**. A leitura usa as instruções **POP**, **POPF**.

3.1.3 Transferencias CPU <> E/S

O 80x86 endereça periféricos de entrada através da instrução **IN** e periféricos de saída via a instrução **OUT**. Todas as transferencias são executadas com o registrador AX (16 bits) , AL ou AH (8 bits) na forma sumariada abaixo:

	Destino	Origem
OUT	REG DX ou END <> E/S	REG AX
IN	REG AX	REG DX ou END<>E/S

```
in al, 06H          in ax,dx
in ax,10DH          in ah,dx

out [si], al        out dx,al
out 3DH,ax          out dx,ax
```

3.1.4 Instruções para transferencias de Blocos

São instruções para serem usadas em "loops", pois assumem que mais de um dado será transferido. Incluem incremento ou decremento de apontadores pré-definidos para os dados.

Transferencia MEM <> MEM

Transferencia de blocos de dados entre regiões da memória. Cada execução da instrução permite a transferencia de blocos com dados de 8 bits(MOVSB), 16 bits (MOVSW), entre regiões da memória.

MEM $_{bloco1}$ < movsb,movsw> MEM $_{bloco2}$
Os Endereços de origem , destino são:
SI: endereço de memória do bloco de origem, localizado no segmento DS.
DI: endereço de memória do bloco de destino, localizado no segmento ES.
Flag D: Pós-incremento (D=0) ou decremento (D=1) automático de SI e DI após cada transferencia. Para movsw SI e DI são atualizados de 2 unidades.

Transferencia MEM <> E/S

Transferencia de blocos de dados entre regiões da memória e portas de E/S. Cada execução da instrução permite a transferencia de blocos com dados de 8 bits (INSB,OUTSB), 16 bits (INSW,OUTSW) ou variável (INS,OUTS), entre uma região da memória apontada pelo registrador DI a porta endereçada pelo registrador DX. DI é incrementado ou decrementado de acordo com o conteúdo do flag D. As instruções INS e OUTS transferem dados cujo comprimento é especificado pelo operando envolvido.

```
insb                ; lê 1 byte da porta endereçada por DX.
outsw               ; Escreve 2 bytes na porta endereçada por DX.
ins     ES:port, DX ; lê da porta endereçada por DX.
```

Exemplo :

```
mov    di, word ptr INICIO
mov    dx, 0230h
insb
```

3.2 Instruções aritméticas, Lógicas e de Manipulação de bits.

O 80X86 possui um rico conjunto de instruções em sua ULA, incluindo as quatro operações aritméticas básicas, funções booleanas (and, or, not, xor) e instruções de deslocamento de bits.

Soma

ADD/ADC operando1, operando2

As instruções acima realizam a soma do operando2 com o operando1 armazenando o resultado no operando1. A principal diferença é que ADC soma o conteúdo de CF aos operandos. Altera os flags.

Subtração

SUB/SBB operando1, opernado2

As instruções acima realizam a subtração do operando2 com o operando1 armazenando o resultado no operando1. A principal diferença é que SBB subtrai o conteúdo de CF dos operandos. Altera os flags.

Incremento

INC reg/mem

Esta instrução realiza o incremento do reg/mem de uma unidade. E equivalente a somarmos 1 ao conteúdo de reg/mem. Altera flags quando for o caso.

Decremento

DEC reg/mem

Esta instrução realiza o decremento do reg/mem de uma unidade.É equivalente a subtraímos 1 ao conteúdo de reg/mem. Altera os flags quando for o caso.

Complemento de 2

NEG mem/reg

Realiza o complemento de 2 da posição de memória ou registrador especificado.

Comparação

CMP operando1, operando2

Realiza a comparação do operando2 com o operando1. Esta instrução é equivalente a uma subtração do operando2 do operando1 sem que o resultado vá para o operando1, ela altera somente os **flags**.

Instruções Lógicas

As instruções lógicas existentes são : **AND**, **OR**, **XOR** e **TEST** e podem ser aplicados entre memória x registrador, registrador x registrador e registrador x memória. Não sendo permitido memória X memória. A única instrução que utiliza um operando é NOT a qual faz a negação do operando.

Instruções de deslocamento e rotação

A rotação de bits em uma palavra pode ser feita através das instruções **ROL**, **ROR**, **RLC** e **RRC**. O deslocamento é feito pelas instruções **SAL**, **SAR**, **SHL** e **SHR**. Algumas instruções permitem a manipulação do conteúdo do indicador de carry. A forma geral das instruções é :

mnemônico argumento, contador

onde :

mnemônico : Qualquer dos descritores de instrução (ROR, SHL, ..).
argumento : Registrador ou endereço de memória.
contador: Número de bits a serem rotacionados ou deslocados. Pode ser a constante 1 ou o conteúdo do registrador **CL**.

3.2 Instruções para controle de Fluxo

Já vimos as instruções para movimentação e aritméticas, agora veremos as instruções para controlar o fluxo de controle de um programa. Sem essas instruções não seria possível escrever um programa.

Desvio Incondicional

Desvio incondicional significa desviar para algum lugar sem que seja necessário testar uma condição previa. Você pede para desviar e ele desvia. No 80X86 temos a instrução **JMP.**

JMP argumento

onde : argumento pode ser um label ou uma posição de memória.

Desvio Condicionais

São desvios realizados após o teste de uma condição. A própria instrução especifica que tipo de teste será realizado. Estes teste consistem em verificar os valores de um ou mais flags ou o registrador CX. Na figura 12 temos uma tabela com as instruções e seus testes.

Mnemônico	Descrição	Flag / Reg	Teste
JNA JNBE	above not below or equal	CF & ZF são 0	CF, ZF
JBE JNA	below or equal not above	CF = 1 ou ZF = 1	
JCXZ	CX é 0	CX = 0	Reg CX
JAE JNB	above o equal not below	CF = 0	CF
JB JNAE	below not above or equal	CF = 1	
JG JNLE	greater not less or equal	ZF = 0 & SF = OF	ZF, SF, OF
JLE JNG	less or equal not greater	ZF = 1 ou SF <> OF	
JGE JNL	greater or equal not less	SF = OF	SF, OF
JL JNGE	less not greater or equal	SF <> OF	
JE JZ	equal zero	ZF = 1	ZF
JNE JNZ	not equal not zero	ZF = 0	
JO	overflow	OF =1	OF
JNO	not overflow	OF = 0	
JS	sign	SF = 1	SF
JNS	not sign	SF = 0	
JP JPE	parity parity even	PF =1	PF
JNP JPO	not parity parity odd	PF = 0	

figura 12: Tabela de desvios condicionais

Loop

Existem instruções que combinam em si as instruções DEC CX e JNZ e facilitam a confecção de loops. (Os loops devem estar dentro do mesmo segmento de código).

loop arg ; decrementa CX. Se CX = 0 desvia para arg.

loope/loopz arg ; decrementa CX. Se CX <> 0 e ZF = 1, desvia para arg.

loopne/loopnz arg ; decrementa CX. Se CX <> 0 e ZF = 0, desvia para arg.

4. Procedimentos (sub-rotinas)

Um bom programador quebra o programa em procedimentos. Podemos fazer isso em linguagem assembler, cada procedimento é definido pelas diretivas PROC e ENDP. O procedimento é chamado pela instrução CALL e o retorno para o programa chamador e dado pela execução da instrução RET.

Freqüentemente procedimentos podem chamar outros procedimentos ou mesmo eles chamarem a si próprios (recursivos), passando ou recebendo parâmetros e alterando os valores das flags.

4.1 Definindo um procedimento: PROC e ENDP.

Cada procedimento é definido pelo uso das diretivas **PROC** e **ENDP**. Como são diretivas, estas não geram código assembly, mas indicam para o montador que se trata de um procedimento sendo o início **PROC** e o fim **ENDP**.

sintaxe:

nome **PROC** tipo (FAR/NEAR)
.
. ; código do procedimento
.
RET

nome **ENDP**

onde :

nome : nome do procedimento (rotina).
tipo : FAR se o procedimento esta em um segmento diferente, senão NEAR (Near é padrão).

A diretiva ENDP marca o fim do procedimento.

Ao final de cada procedimento devemos colocar a instrução RET para que o procedimento retorne ao programa que o chamou.

Outro ponto importante : Antes de digitar um procedimento, é necessário colocar um comentário na forma de cabeçalho, onde se explica o que o procedimento realiza. Abaixo temos um exemplo, o qual deve ser seguido em todos os programas onde se utilize procedimentos.

;**********

```
; Nome :
;
; Objetivo :
;
; Descrição :
;
; Entradas :
;
; Saídas :
;
; Destroi :
;
; Data :
;
; Autor :
;
;*****************************************************
```

4.2 Chamando um procedimento

Para chamarmos um procedimento utilizamos uma instrução chamada CALL, e podemos utilizá-la de duas maneiras:

CALL DISPLAY - onde chamamos o procedimento pelo seu nome, e o montador resolve o seu endereço.

CALL table[BX] - onde usamos um endereço indexado.

4.2.1 Como funciona a instrução CALL

A instrução CALL realiza as seguintes tarefas:

- empilha o endereço de retorno;
- desvia para o endereço do procedimento

Obs.: se o procedimento for FAR - empilha-se o CS + IP (endereço de retorno).

4.3 Terminando um procedimento

Ao final de um procedimento, este deve ser encerrado pela instrução RET.

4.3.1 Como funciona a instrução RET.

Como a instrução CALL empilha o endereço de retorno, devemos ter uma instrução que retire esse endereço de retorno da pilha, e essa instrução é a RET.
Ela retira da pilha o endereço de retorno, o qual é composto pelo IP ou CS+IP se for FAR.

4.4 Passando/recebendo parâmetros para um procedimento.

Existem duas maneiras de passar/receber parâmetros de um procedimento.

- A primeira consiste no uso dos registradores de uso geral, podemos passar quanto receber parâmetros através deles. Devemos tomar cuidado se desejamos preservar os seus valores antes de chamar o procedimento, e também devemos especificar a sua utilização no cabeçalho do procedimento.

- O segundo meio é através da pilha. Este método é um pouco mais complexo, mas evita a destruição dos registradores e podemos passar ou receber quantos parâmetros quisermos.

Como funciona? Antes de chamarmos o procedimento empilhamos os parâmetros na pilha. Chamamos o procedimento. No procedimento fazemos com que BP se iguale ao SP. (Não utilizamos SP diretamente, pois erros podem ser desastrosos).
Estando BP apontando para o topo da pilha, o que devemos fazer é localizar os parâmetros na pilha, para isso somamos ao conteúdo de BP o deslocamento do parâmetro.

BP + 2 -> 1º parâmetro
BP + 4 -> 2º parâmetro
e assim por diante.

Esse deslocamento só é válido se o procedimento for NEAR, caso ele seja FAR devemos começar com BP + 4 para o primeiro parâmetro e assim por diante. (Isso por que na pilha alem do IP está o CS e este ocupa 2 bytes).

Exemplo :

```
mov    [BP + 2], AX          ; passa o valor de ax para a pilha

mov    AX, [BP + 4]          ; carrega AX com o valor do parâmetro
```

5. Interrupções

O processador deve ser capaz de enfrentar vários eventos , os quais podem ocorrer a qualquer momento. A maioria desses eventos são originados pelo hardware. Alguns deles são :

- Um sinal do teclado dizendo que uma tecla foi pressionada.
- Um sinal da impressora dizendo que está sem papel.
- Um sinal do acionado de disco informando que terminou um transferência de dados.
- Um chip de memória ruim detectado.

Ocasionalmente um evento não previsto pode ocorrer no software:

- Um programa tentou fazer uma divisão por zero.

A questão é , como o processador irá tratar esses eventos sem interferir no trabalho que está sendo executado? Imagine você em uma situação parecida. Suponha que você está na cozinha, seguindo uma receita para preparar um coelho ensopado. Em vários momentos imprevisíveis, você é distraído por outros eventos: alguém bate na porta, o telefone toca, você corre atras do coelho, e outros. De qualquer modo, você deve enfrentar esta distrações sem misturar os passos da receita.

Geralmente falando, existem duas maneiras de enfrentar está situação. A primeira, você pode ignorar os eventos externos assim que eles acontecem, mas fazer uma pausa em intervalos de tempo regulares para ver se alguma coisa precisa de sua atenção. A segunda. você pode lidar com os eventos externos assim que eles acontecem. Entretanto, você deve cuidadosamente marcar o lugar da receita que você parou para que possa continuar depois no ponto certo onde parou.

O processador é projetado para usar o segundo sistema. A todo tempo, o computador é preparado para reconhecer um sinal o qual indica se um evento externo ocorreu, e que necessita de sua atenção imediatamente. Este sinal é chamado **Interrupção**.

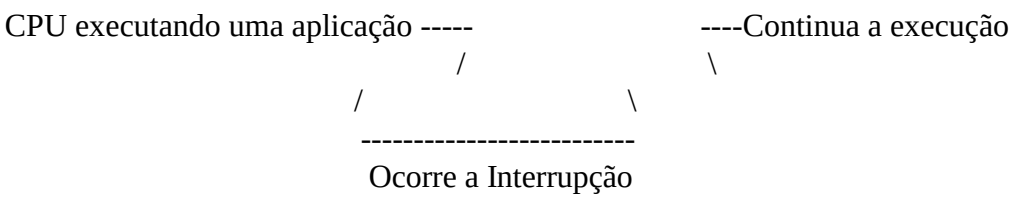
Quando uma interrupção ocorre, o processador cuidadosamente põe de lado o programa corrente, trata a interrupção, e retorna a executar o programa que tinha parado. Podemos dizer que o processador servio a interrupção. Embora possa parecer que seja uma operação complicada, o processador executa tão rápido que a interrupção pode ser conhecida e servida em uma pequena fração de tempo.

Interrupções podem ocorrer a qualquer momento, e o processador deve ter dois recursos básicos para trata-las:

1 - O processador deve poder desabilitar o recebimento de interrupções.

2 - Quando ocorrer um interrupção, o processador deve ser capaz de salvar as informações sobre o que está fazendo. Isto é necessário para que ele possa retornar ao ponto anterior assim que a interrupção tenha sido tratada.

A seguir , esquematizamos uma interrupção:



5.1. Como são servidas as interrupções.

Após detectar uma interrupção, o processador executa um procedimento especial chamado "tratador de interrupção". Cada tipo de interrupção requer seu próprio tratador de interrupção. O propósito do tratador de interrupção é fazer o que for necessário para cuidar das causas da interrupção. Uma vez que o tratamento da interrupção termina, o processador continua com o programa que estava sendo executado.

Do ponto de vista do processador, ao começar o tratamento de um interrupção é parecido como começar uma chamada de procedimento. A principal diferença é que antes de desviar a execução para o tratamento de interrupção, o processador deve salvar informação suficiente para poder voltar a executar o programa do ponto que ele parou. Em particular, o processador deve salvar as flags e o endereço completo (CS e IP) da próxima instrução do programa que está sendo executado.

Aqui há um sumário dos passos que o processador lida quando lida com uma interrupção:

1. Termina a instrução que está sendo executada.
2. Coloca na pilha o registrador flag (pushf)
3. Coloca na pilha o endereço do segmento da próxima instrução.
4. Coloca na pilha o offset da próxima instrução.
5. Determina o endereço apropriado do manipulador da interrupção.
6. Começa a executar o manipulador da interrupção.

Quando o manipulador de interrupção termina, o processador restaura o flag e recomeça o programa que havia interrompido.

5.2. Descritor da tabela de interrupção

Cada interrupção é servida pelo seu próprio manipulador de interrupção. Quando uma interrupção acontece, o processador deve ser capaz de identificar o tipo de interrupção e determinar o endereço apropriado para o manipulador de interrupção.

O sistema é desenvolvido para que cada diferente interrupção seja identificada por um número chamado de vetor de interrupção. O vetor de interrupção pode ser escrito em decimal ou hexadecimal. Para evitar confusão, nos sempre usaremos escrever o vetor de interrupção em hexadecimal.

O DOS mantém a tabela de endereços completos, cada um apontando para um diferente manipulador de interrupção. Esta tabela é chamada de Descritor da tabela de interrupção. O Descritor é armazenado na mais baixa posição da memória, de 0000H a 03FFH(1024 bytes). Desde que cada endereço completo (segmento e offset) requerem 4 bytes, o descritor pode manter 256 (1024/4) endereços completos. Isto significa que o DOS possui nele a facilidade para servir 256 interrupções diferentes, embora muitas delas não são utilizadas.

As 256 possíveis interrupções são numeradas de 0H a FFH (0 a 256). Os endereços para as interrupções são armazenados na mesma ordem no descritor. Assim, o endereço completo para o manipulador da interrupção de número 0 é armazenado no quatros bytes começando na localização 0000H. O endereço para a 1 é armazenado na localização 0004H. O endereço para a 2 é colocado na posição 0008H. Em geral, o endereço para a enezima interrupção é a posição 4 x n.

A tabela de descritores é carregada automaticamente assim que se inicia o DOS.

O 1K de memória, da posição 0000:0000 até 0000:03FF, é reservado para guardar o endereço dessas rotinas. Cada endereço é composto de 4 bytes, sendo que as duas primeiras posições guardam o offset e as duas seguintes o segmento de memória.

A seqüência de tratamento de uma interrupção qualquer seria:

1 - Os Flags são colocados no stack.

2 - O CS é colocado no stack.

3 - O IP é colocado no stack.

4 - O 8088 determina o tipo de interrupção.

5 - A CPU determina o endereço do endereço (pointer) da rotina que irá tratar essa interrupção.

6 - Os dois primeiros bytes são colocados no IP.

7 - Os dois seguintes são colocados no CS.

8 - A execução é transferida para o endereço CS:IP

9 - A rotina de tratamento de interrupção é executada.

10 - No Final da rotina, é executada a instrução IRET.

11 - O IP é restaurado.

12 - O CS é restaurado.

13 - Os flags são retirados do stack.

14 - A execução continua do ponto em que parou.

Vejamos um exemplo:

Suponhamos que o endereço de memória 0000:0030 contenha a seqüência 1F 2C 00 F0. Quando o 8088 detecta una interrupção, multiplica o numero desta interrupção por 4, e o valor encontrado será o endereço de onde está armazenado o endereço da rotina de tratamento para aquela interrupção. Por exemplo, a interface serial gera uma interrupção do tipo 12 (se for a COM1). O endereço do pointer dessa rotina estará, então, no endereço 12 * 4 = 48. Convertendo-se esse endereço em hexadecimal, obteríamos o endereço 0000:0030H. O conteúdo desse endereço seria colocado em IP e em CS, obtendo assim IP = 2C1F e CS = F000, e a execução seria transferida para este ponto.

Como já foi visto, o 8088 tem até 256 tipos diferentes de interrupção, mas existem algumas(mais exatamente 5 delas) que são geradas pelo próprio processador.

Interrupção 0, endereço 0000:0000: é acionada quando for realizada uma divisão por 0, utilizando as instruções DIV ou IDIV.

Interrupção 1, endereço 0000:0004: gerada após a execução de uma instrução qualquer se o TRAP FLAG estive em 1. (SINGLE STEP INTERRUPT), é utilizado muito com os depuradores.

Interrupção 2, endereço 0000:0008: interrupção não-mascarável (NMI- Nom Maskable INTERRUPT), é acionada quando ocorrer um erro de paridade na memória, possuindo esse nome por não existir um meio de, através do software, desabilita-la.

Interrupção 3, endereço 0000:000C : breakpoint. Utilizado pelos depuradores para colocar breakpoints.

Interrupção 4, endereço 0000:0010 : overflow, esta interrupção ocorre quando o flag de overflow (OF) estiver ativado e for executado a instrução INTO. Isso ocorre quando fizemos uma operação que exceda o tamanho do registrador destino. Por Exemplo, se somarmos FFFF com FFFF, certamente teremos um overflow; executando-se a instrução INTO,a execução do programa será transferida para a rotina apontada pelo endereço 0000:0010. Normalmente esse endereço aponta para uma posição de memória que contem somente IRET

5.3 .Como uma interrupção é utilizada por um programador : INT

Interrupção possuem dois propósitos: O primeiro, eles permitem que o computador responda quase que imediatamente a eventos que acontecem aleatoriamente. Na maior parte, estes eventos são causados por sinais de vários dispositivos de hardware. Assim, a interrupção é o modo do computador permitir o processador suportar o hardware. Entretanto, interrupções são usadas para um propósito completamente diferente.

Se você está pensando sobre isto, você vera que o mecanismo de interrupção permite que o processador chame um procedimento que ele apenas o conhece por um número particular (Vetor de interrupção). Se é possível para um programa sinalizar uma interrupção, o programa deve ser capaz de chama-la sem precisar saber seu nome ou localização.

Por que deveria um programa necessitar esta chamada como um procedimento? A resposta é que o sistema operacional (DOS) e os software build-in (BIOS) oferecem vários serviços para os programas. E para o programa obter vantagem desses serviços, o programa deve poder chamar os procedimentos apropriados. Entretanto, não existe um modo do programa saber a localização deste procedimentos.

A solução é atribuir para cada serviço um número que corresponde ao vetor de interrupção. Um programa poder requisitar o serviço sinalizando a interrupção apropriada. Em fato, existe uma instrução, INT (interrupt), especialmente para este propósito. O seu formato é :

```
INT         vetor-interrupção
```

Por exemplo, um programa pode sinalizar a interrupção 21H pela execução da seguinte linha:

```
int         21h
```

Assim, Interrupções tem dois propósitos:

- Permitir que o processador responda a necessidades do hardware;
- Permitir que o programa requisite um serviço do sistema operacional ou do software que acompanha o equipamento (built-in).

Uma interrupção que se origina de um dispositivo de hardware é chamada de "hardware interrupt". A interrupção que se origina de um programa que está sendo executado é chamada de "software interrupt". Nós raramente iremos nos concentrar em interrupções de hardware. Servir uma interrupção de hardware é algo complicado e devemos deixa-las com a BIOS. Os programador de assembler devem se ater mais as interrupções de software.

5.4. As Chamadas de Função do DOS.

DOS provem mais de 130 serviços úteis para o programa. Ao invés de dar a cada serviço, o seu próprio número de interrupção, o DOS usa um número geral, o 21H. Os serviços que o DOS oferece e que estão sob os cuidados da interrupção 21H, são chamados de chamadas de função.

Cada chamada de função é identificada por um número chamado de número de chamada. Para usar a chamada de função, coloca-se no registrador AH o número, então aciona-se a interrupção 21H. Por exemplo, a chamada 19H determina qual é o drive default.

```
: determina o drive default corrente
  mov    ah,19h
  int    21h
```

Desde que todas as chamadas de função são executadas pela mesma interrupção, ela provem uma padronização no modo de oferecer serviços para o programador.

Glossário:

Interrupção : Um sinal de um evento que ocorreu e que precisa ser imediatamente atendido.

Servir : Manipular uma interrupção.

Manipulador de interrupção : Um procedimento que executa qualquer coisa necessária quando uma interrupção acontece.

Vetor de interrupção : Um número que identifica uma interrupção especifica.

Tabela de descritores de interrupção : Uma tabela com os endereço completos, cada um dos quais aponta para um manipulador de interrupção.

Interrupção de Hardware : Uma interrupção originada a partir de um dispositivo de hardware. Também conhecida com interrupção externa.

Interrupção de Software : Um interrupção que é originado a partir de um programa em execução. Também conhecida como interrupção interna.

Function Call (Chamada de função) : Um serviço oferecido pelo DOS sob o patrocínio da interrupção 21H.

Call number (Numero de Chamada) : O número que identifica uma "Function Call" especifica.

Program segment Prefix (PSP) : Uma área de 100H(256) bytes de memória que contem um conjunto de informações usadas pelo DOS para executar um programa.

Bibliografia:

Guia do Programador IBM PC - Peter Norton
Editora Campus

Programando em assembler 8086/8088 - Jeremias Pereira dos Santos, Edison Raymundi Jr.
Editora McGraw-Hill.

Assembler Inside e OUT - Harley Hahn
Osborne - McGraw-Hill

Manual do microprocessador 8086/88 - INTEL