





MICROPROCESSADORES

CONCEITOS IMPORTANTES

– 2.^a EDIÇÃO –

Roberto M. Ziller
– Edição do autor –
Florianópolis - SC - Brasil
2000



Copyright © 1999, 2000 Roberto M. Ziller

Direitos autorais reservados.

Ficha catalográfica:

Z69m Ziller, Roberto M.
Microprocessadores: conceitos importantes / Roberto M.
Ziller . – 2. ed. – Florianópolis : Ed. do autor, 2000.
256 p. : il., tabs.

Inclui bibliografia.

ISBN 85-901037-2-2

1. Microprocessadores. 2. Intel 8085 (Microprocessador)
3. Intel 8086 (Microprocessador). I. Título.
CDU: 621.38.037-181.4

Catálogo na fonte por: Onélia Silva Guimarães CRB-14/071

Agradecimentos

Agradeço ao professor Vitório Bruno Mazzola, do Departamento de Informática e Estatística da UFSC, pelo material cedido, que serviu de inspiração para alguns capítulos, ainda na primeira edição deste trabalho.

Agradeço especialmente ao professor Werner Kraus Jr., do Departamento de Automação e Sistemas, pela revisão crítica e cuidadosa do texto e pelas inestimáveis sugestões que ajudaram a torná-lo mais claro e mais rico. Assim como seu colega Joni da Silva Fraga, incentivou-me muitas vezes a inovar e melhorar o material didático.

Agradeço também aos professores Hari Bruno Mohr e Raimes Moraes, colegas do Departamento de Engenharia Elétrica, pela constante disposição para discutir detalhes técnicos e pelo incentivo que tornou realidade este livro.

Ao professor Geraldo Kindermann, também colega de departamento, devo um agradecimento especial, por ter colocado à minha disposição toda a sua experiência como autor e editor, mostrando-me o melhor caminho para realizar esta publicação. O mesmo fez o professor e colega Arnaldo José Perin, quando da publicação da edição anterior.

Aos acadêmicos Rafael Carlos Jung Sperotto e Thiago Pereira Berto, do curso de Engenharia de Controle e Automação, e Marcelo Luís L. Santos, do curso de Engenharia Elétrica, agradeço a busca de erros na primeira edição. Aos meus alunos de Microprocessadores do segundo semestre de 1999 agradeço a dedicação e o empenho que demonstraram nos testes do Abacus.

Agradeço também o apoio recebido do Departamento de Engenharia Elétrica e do Departamento de Automação e Sistemas.

De forma coletiva, agradeço as sugestões, as críticas e o incentivo de muitos outros alunos, colegas e amigos.

À Diomara agradeço o carinho, o amor e a paciência, sem os quais este trabalho não teria sido terminado.



Apresentação

A velocidade com que acontece a evolução da Informática poderia levar a crer que a tarefa do professor de uma disciplina como a de Microprocessadores seria a de atualizar continuamente o material apresentado aos alunos, para que estes ficassem a par das constantes inovações tecnológicas.

Esta idéia não resiste, porém, a uma análise mais profunda: se é este o tipo de conhecimento dado aos alunos que estão no meio do seu curso, então dificilmente ele lhes será útil quando se formarem, depois de alguns anos. Os conceitos fundamentais que norteiam o funcionamento dos microprocessadores são, sob este aspecto, mais importantes do que os últimos lançamentos da indústria, porque não mudam e serão úteis sempre. Por esta razão, o material aqui contido procura valorizar, sobretudo, os conceitos independentes da inovação tecnológica. Da mesma forma, justifica-se a utilização do microprocessador 8085 nos primeiros capítulos, pois este apresenta a simplicidade necessária ao início dos estudos, ao mesmo tempo que os conhecimentos adquiridos são empregados no restante do livro, que utiliza o 8086. Este corresponde aos processadores mais avançados da Intel quando operam no modo real e serve, assim, de base para um estudo mais aprofundado.

Acompanhando o trabalho dos alunos que concluem a disciplina, pude constatar que eles são capazes de entender o funcionamento de microprocessadores diferentes daqueles que estudamos e de trabalhar com eles, aplicando os princípios que aprenderam. É por isso que acredito nesta abordagem, e foi também este o principal fator que me levou a aceitar diversas sugestões de alunos e colegas para lançar este livro.

Apesar dos novos capítulos acrescentados nesta edição, o texto não pretende cobrir de forma completa o vasto assunto de que trata e não dispensa a bibliografia citada. Deve ser entendido como um guia para aqueles que pretendem se aprofundar em livros específicos sobre um ou outro processador, em que já não há espaço para tratar os conceitos mais básicos. A densidade da

exposição aumenta ao longo do texto, tornando-o indicado para um curso de graduação. À medida que o aluno avança em sua leitura, passa a receber informações num estilo cada vez mais próximo daquele dos manuais e livros que terá de ler quando trabalhar em um projeto profissional.

O livro tem uma parte teórica e outra, prática, cujo estudo pode ser iniciado simultaneamente. Neste caso, o professor de laboratório tem que adiantar alguns conceitos antes que sejam apresentados na teoria, mas isso é necessário apenas nas primeiras aulas, pois logo ambas as partes se sincronizam. O nível de detalhe das aulas de laboratório não precisa ser o mesmo das aulas teóricas, e este pequeno esforço adicional tem sido recompensado por um ganho de tempo significativo em cada semestre e pela maior motivação dos alunos para o estudo completo da teoria relacionada.

Alternativamente, pode-se avançar primeiro na teoria até o final do capítulo sobre conceitos básicos, e iniciar a parte de laboratório juntamente com o estudo teórico do 8085. Esta abordagem é a mais interessante para o autodidata ou para aqueles que ainda não tiveram contato com o material do capítulo 2, que normalmente é uma revisão de conhecimentos já adquiridos em outras disciplinas.

As novidades desta edição são os capítulos sobre interrupções e manipulação de strings do 8086 e, principalmente, a introdução do software Abacus, desenvolvido especialmente para acompanhar o livro e permitir a realização de todos os experimentos com o 8085 em um computador pessoal. As facilidades incluídas neste programa baseiam-se em vários anos de experiência em testes e depuração de software e na observação das principais dificuldades encontradas pelos alunos na visualização dos conceitos mais importantes.

O texto aproveitado da primeira edição sofreu uma revisão completa, e os exercícios considerados mais difíceis receberam mais dicas para orientar sua solução.

Espero que o esforço dedicado a estas modificações possa melhorar um pouco mais a minha modesta contribuição para o estudo dos microprocessadores.

O Autor

Florianópolis, 07/02/2000.



À minha mãe





Abacus – apoio via Internet

A página www.eel.ufsc.br/microprocessadores oferece amplo suporte ao material aqui contido. A partir deste endereço, é possível obter o Abacus, um simulador para o microprocessador 8085, desenvolvido pelo autor. O Abacus é um programa para o ambiente Microsoft Windows, que permite executar os programas para o 8085 contidos no livro e também resolver os exercícios propostos. Desta forma, o aluno que tenha seu próprio computador não precisa depender de horários de laboratório para realizar os experimentos, e a instituição de ensino pode utilizar melhor o potencial representado pelos computadores de seus alunos.

Também o autodidata e o profissional mais experiente podem se beneficiar do uso do Abacus, uma vez que este se constitui numa ferramenta de simulação e depuração de praticamente qualquer programa escrito para o 8085.

Além do Abacus, a página citada permite obter as listagens dos programas contidos no livro e oferece outras informações interessantes, tais como links para páginas com informações complementares e bibliografia.

O Abacus é gratuito sempre que utilizado para fins educacionais.



Emprego de termos técnicos

O presente texto faz amplo uso de diversos termos da Informática e da Engenharia Elétrica. Alguns desses termos, embora de uso corrente no jargão técnico, não fazem parte do vocabulário oficial da Língua Portuguesa e por isso deveriam, a rigor, aparecer em destaque no texto. Entretanto, considerando a naturalidade com que tais termos aparecem na comunicação oral e na comunicação informal escrita entre pessoas dessas áreas, optou-se por escrevê-los de forma indistinta, em benefício da estética e da uniformidade, já que o destaque dessas palavras apenas devido à sua origem poderia desviar desnecessariamente a atenção do leitor.

A seguir, uma lista dos principais termos que recaem nessa classificação e seu significado:

assemblar:	gerar o código objeto correspondente a um programa;
assembler:	ferramenta que gera o código objeto de um programa;
bit:	dígito binário;
byte:	palavra de 8 bits;
carry:	variável binária que sinaliza “vai 1” ou “empresta 1” em operações aritméticas;
chip:	pastilha circuito integrado;
debugger:	ferramenta para depurar programas;
default:	diz-se do valor assumido por uma variável na ausência de valor especificado;
display:	mostrador, tipicamente digital, de leds ou cristal líquido;
doubleword:	conjunto de duas palavras de 16 bits;
driver:	dispositivo de hardware ou software;
flag:	variável binária;
label:	rótulo ou etiqueta que identifica uma linha de código;
latch:	circuito digital de armazenamento;

linkar:	gerar a forma final de um programa a partir dos módulos com código-objeto;
linker:	ferramenta para linkar;
on-chip:	na mesma pastilha de circuito integrado;
opcode:	código hexadecimal que identifica uma operação;
pointer:	apontador;
prompt:	espaço da interface de usuário de um programa em que se digitam comandos;
quadword:	palavra de 64 bits;
reset:	volta ao estado inicial ou atribuição do valor 0 a uma variável;
set:	atribuição do valor 1 a uma variável;
stack:	pilha;
string:	cadeia de caracteres alfanuméricos;
timer:	temporizador;
word:	palavra de 16 bits.

X8085 e LINK2 são programas da 2500 A.D. SOFTWARE, INC.

MASM, LINK, Symdeb, MS-DOS e Windows são programas da Microsoft.

Sumário

AGRADECIMENTOS	III
APRESENTAÇÃO	V
ABACUS – APOIO VIA INTERNET	IX
EMPREGO DE TERMOS TÉCNICOS	XI
SUMÁRIO	XIII
LISTA DE FIGURAS	XXI
LISTA DE TABELAS.....	XXIII
PARTE I – TEORIA	25
1 INTRODUÇÃO AOS COMPUTADORES.....	27
1.1 OBJETIVO.....	27
1.2 UM POUCO DE HISTÓRIA	28
1.2.1 A primeira geração (1941 – 1955)	29
1.2.2 A máquina de von Neumann (1949)	30
1.2.3 A segunda geração (1955 – 1965)	31
1.2.4 A terceira geração (1965 – 1980).....	31
1.2.5 A quarta geração (1980 – ...)	32
1.3 A FAMÍLIA INTEL	32
2 CONCEITOS BÁSICOS	35
2.1 SISTEMAS DE NUMERAÇÃO	35
2.2 CONVERSÃO DE BASE	37
2.2.1 Casos particulares de conversão de base.....	39

xiv	Microprocessadores: conceitos importantes	
2.3	REPRESENTAÇÃO DE NÚMEROS INTEIROS.....	40
2.3.1	Obtenção do valor simétrico de um número	42
2.3.2	Subtração usando adição	43
2.4	A REPRESENTAÇÃO BCD	43
2.5	TABELAS DE CODIFICAÇÃO DE CARACTERES	46
2.6	MANIPULAÇÃO DE BITS	48
2.7	TIPOS DE MEMÓRIAS	49
2.8	OPERAÇÕES DESTRUTIVAS E NÃO DESTRUTIVAS	50
2.9	INTERPRETAÇÃO DO CONTEÚDO DA MEMÓRIA	51
2.10	A EXECUÇÃO DE PROGRAMAS	51
2.10.1	A execução de uma instrução	52
2.11	A MEMÓRIA E OS BARRAMENTOS	53
3	INTRODUÇÃO AO 8085	57
3.1	ORIGEM E CARACTERÍSTICAS TÉCNICAS.....	57
3.2	O MODELO DE PROGRAMAÇÃO.....	60
3.3	A PILHA	62
3.4	SUB-ROTINAS.....	64
3.4.1	O papel da pilha na chamada de sub-rotinas.....	64
3.4.2	Utilização da pilha na preservação de registradores	66
4	PROGRAMAÇÃO DO 8085	67
4.1	ASSEMBLER E LINGUAGEM ASSEMBLY	67
4.2	EMPREGO DA LINGUAGEM ASSEMBLY	68
4.3	INSTRUÇÕES DO 8085	68
4.3.1	Formato	69
4.3.2	Classificação	70
4.4	FORMATO DE UMA LINHA DE CÓDIGO ASSEMBLY	70
4.5	DIRETIVAS DO ASSEMBLER.....	71
4.6	ETAPAS DO DESENVOLVIMENTO EM ASSEMBLY	73
4.7	EXEMPLO DE PROGRAMA EM ASSEMBLY	73
4.8	OUTROS TRECHOS DE CÓDIGO.....	76
4.8.1	Tomando decisões.....	76

4.8.2 Repetições.....	79
4.8.3 Casos particulares de multiplicação	79
5 INTERRUPÇÕES DO 8085	81
5.1 INTERRUPÇÕES X SUB-ROTINAS.....	81
5.2 HABILITANDO E INIBINDO INTERRUPÇÕES.....	83
5.3 PRIORIDADES DAS INTERRUPÇÕES.....	86
5.4 INTERRUPÇÕES DE HARDWARE E DE SOFTWARE	86
5.5 ESCRIVENDO TRATADORES DE INTERRUPÇÃO	87
6 O MICROPROCESSADOR 8086.....	89
6.1 INTRODUÇÃO	89
6.2 8086 X 8088	89
6.3 UM PEQUENO PROBLEMA	90
6.4 SEGMENTAÇÃO	91
6.4.1 Notação.....	92
6.4.2 Multiplicidade de endereços.....	93
6.5 OS REGISTRADORES DE SEGMENTO	93
6.6 A VISÃO DO PROCESSADOR.....	94
6.7 O MODELO DE PROGRAMAÇÃO.....	95
6.8 CONSIDERAÇÕES SOBRE A LINGUAGEM ASM-86	96
6.9 FORMATO DAS INSTRUÇÕES DO 8086.....	97
6.10 MODOS DE ENDEREÇAMENTO	98
6.10.1 Endereçamento via registrador	99
6.10.2 Endereçamento imediato	99
6.10.3 Endereçamento absoluto ou direto	100
6.10.4 Endereçamento indireto.....	100
6.10.5 Endereçamento indexado	101
6.10.6 Endereçamento baseado	102
6.10.7 Endereçamento baseado indexado.....	104
6.10.8 Endereçamento relativo	104
6.10.9 Determinação do segmento utilizado.....	106

xvi	Microprocessadores: conceitos importantes
7	PARÂMETROS E VARIÁVEIS LOCAIS..... 109
7.1	INTRODUÇÃO 109
7.2	PASSAGEM DE PARÂMETROS PARA SUB-ROTINAS..... 109
7.2.1	Passagem através de registradores..... 109
7.2.2	Passagem através de variáveis globais 110
7.2.3	Passagem através da pilha 111
7.2.4	Aninhamento de sub-rotinas (nesting) 114
7.2.5	Comparação entre os métodos..... 116
7.3	CRIAÇÃO DE VARIÁVEIS LOCAIS 117
7.3.1	Variáveis locais com inicialização..... 120
8	INTERRUPÇÕES DO 8086121
8.1	INTERRUPÇÕES..... 121
8.1.1	Características gerais 121
8.1.2	A tabela de vetores de interrupção 121
8.1.3	A sinalização das interrupções..... 123
8.1.4	Habilitando e desabilitando..... 123
8.1.5	O desvio para o tratador..... 124
8.1.6	Interrupções de software..... 124
8.1.7	Interrupções reservadas 125
9	MANIPULAÇÃO DE STRINGS..... 127
9.1	INTRODUÇÃO 127
9.2	AS INSTRUÇÕES 127
9.3	UM EXEMPLO – CÓPIA DE TABELAS..... 129
9.4	OS PREFIXOS DE REPETIÇÃO..... 131
9.5	OCORRÊNCIA DE INTERRUPÇÕES..... 132
9.6	EXEMPLOS DE APLICAÇÃO 133
9.6.1	Comparação de tabelas 133
9.6.2	Varredura de tabelas..... 134
9.6.3	Leitura de tabelas..... 134
9.6.4	Inicialização de tabelas..... 135

PARTE II – LABORATÓRIO	137
10 O ABACUS	139
10.1 O SIMULADOR E O ASSEMBLER.....	139
10.2 A MEMÓRIA	140
10.3 AS JANELAS.....	140
10.3.1 A janela Programa	141
10.3.2 A janela Processador.....	143
10.3.3 A janela Memória	144
10.3.4 A janela Display.....	145
10.3.5 A janela Leds.....	145
10.3.6 A janela Chaves	145
10.3.7 A janela Assembler.....	146
10.4 OS MENUS	147
10.4.1 O menu Arquivo	147
10.4.2 O menu Ferramentas	148
10.4.3 O menu Opções	148
10.4.4 O menu Ajuda	150
10.5 CODIFICAÇÃO DE INSTRUÇÕES DO 8085	151
10.6 O ASSEMBLER DO ABACUS	152
10.7 EXERCÍCIOS.....	153
11 FERRAMENTAS DE DESENVOLVIMENTO	155
11.1 O PAPEL DO ASSEMBLER	155
11.2 UTILIZAÇÃO DAS FERRAMENTAS DE SOFTWARE.....	157
11.2.1 Edição	158
11.2.2 Montagem.....	158
11.2.3 Linkagem	159
11.2.4 Execução	160
11.3 EXEMPLO.....	160
11.4 EXERCÍCIOS.....	161
12 SUB-ROTINAS SIMULADAS EM ROM.....	163
12.1 AS SUB-ROTINAS	163

xviii	Microprocessadores: conceitos importantes	
12.2	EXEMPLO	164
12.3	EXERCÍCIOS	165
13	DATA JULIANA	169
13.1	OBJETIVO	169
13.2	EXERCÍCIO PROPOSTO	169
13.3	SOLUÇÃO	171
13.4	EXERCÍCIOS COMPLEMENTARES	172
14	CONVERSÃO DE BASE.....	173
14.1	LISTAGEM DE HEXCONV.ASM.....	174
14.2	LISTAGEM DE BASECONV.ASM.....	177
14.3	EXERCÍCIOS COMPLEMENTARES	180
15	INTERRUPÇÕES NA PRÁTICA.....	181
15.1	UTILIZAÇÃO DAS TECLAS DE INTERRUPÇÃO DO ABACUS	181
15.2	TRATADORES DE INTERRUPÇÃO NA PRÁTICA	182
15.2.1	Contador decimal com inibidor por interrupção.....	182
15.2.2	Como não fazer.....	183
15.2.3	Como fazer.....	187
15.3	EXERCÍCIOS.....	188
16	PORTAS DE ENTRADA E SAÍDA.....	191
16.1	AS PORTAS DE E/S DO ABACUS	191
16.2	A LIGAÇÃO DO 8155 AO 8085	192
16.3	CONFIGURAÇÃO DO 8155	192
16.4	EXEMPLO E EXERCÍCIOS.....	193
17	ASSEMBLY PARA O PC	195
17.1	O ASSEMBLER.....	195
17.2	O LINKER.....	196
17.3	O DEBUGGER.....	196
17.3.1	Comandos.....	197
17.4	EXERCÍCIOS.....	197

Sumário	xix
18 TESTE E DEPURAÇÃO	201
18.1 SUB-ROTINAS NEAR E FAR	201
18.1.1 Explorando as listagens	202
18.1.2 Explorando os recursos do Symdeb.....	203
18.1.3 Introduzindo erros	205
18.1.4 Passando parâmetros	206
19 ENDEREÇAMENTO BASEADO E INDEXADO	207
19.1 TABELAS E MATRIZES.....	207
19.2 EXERCÍCIOS.....	208
20 DESVIANDO INTERRUPÇÕES	211
20.1 PROGRAMAS RESIDENTES.....	211
20.2 DOS IDLE INTERRUPT	212
20.3 DOS IDLE INTERRUPT E O WINDOWS	212
20.4 O PROGRAMA.....	213
20.5 EXERCÍCIOS.....	214
21 TRABALHANDO COM STRINGS.....	215
21.1 INICIALIZAÇÃO E CÓPIA DE TABELAS.....	215
21.2 EXERCÍCIOS.....	216
21.3 VARREDURA DE STRINGS	217
21.4 EXERCÍCIOS.....	219
ANEXO 1 – INSTRUÇÕES DO 8085.....	221
ANEXO 2 – INSTRUÇÕES DO 8086.....	225
2.1 ABREVIATURAS, FLAGS E DIRETIVAS	225
2.2 TIPOS	226
2.3 AS INSTRUÇÕES.....	226
ANEXO 3 – COMANDOS DO SYMDEB	229
ANEXO 4 – SERVIÇOS DO DOS E DO BIOS.....	233
4.1 FUNÇÕES DO BIOS	233
4.1.1 Serviços de vídeo (INT 10H).....	233
4.2 FUNÇÕES DO DOS.....	236

xx	Microprocessadores: conceitos importantes	
	4.2.1 Serviços gerais (INT 21H).....	236
	4.2.2 DOS Idle interrupt (INT 28H).....	237
	ANEXO 5 – ARQUIVOS .COM	239
	ANEXO 6 – A DIRETIVA ASSUME	241
	ANEXO 7 – PROGRAMAS COM MÚLTIPLOS ARQUIVOS	245
	ANEXO 8 – DIVISÃO DE NÚMEROS INTEIROS	249
	REFERÊNCIAS BIBLIOGRÁFICAS	253
	OUTROS LIVROS DE PROFESSORES DO EEL / UFSC	255

Lista de figuras

FIG. 1.1 – MODELO DA MÁQUINA DE VON NEUMANN	30
FIG. 2.1 – O ALGORITMO DE CONVERSÃO DE BASE.....	38
FIG. 2.2 – REPRESENTAÇÃO CIRCULAR DOS NÚMEROS DE 4 BITS	40
FIG. 2.3 – REPRESENTAÇÃO DE NÚMEROS POSITIVOS E NEGATIVOS	41
FIG. 2.4 – RELAÇÕES DE COMPLEMENTO ENTRE SIMÉTRICOS	42
FIG. 2.5 – OS DÍGITOS DO SISTEMA HEXADECIMAL E UM ATALHO.....	44
FIG. 2.6 – A SOMA DE NÚMEROS BCD.....	45
FIG. 2.7 – ALGUNS ELEMENTOS DA CPU E DA MEMÓRIA.....	52
FIG. 2.8 – CONEXÃO DE UM MICROPROCESSADOR À MEMÓRIA	54
FIG. 2.9 – MEMÓRIA DE 64 kB.....	55
FIG. 3.1 – DISTRIBUIÇÃO DOS SINAIS NO CIRCUITO INTEGRADO 8085.....	57
FIG. 3.2 – MODELO DE PROGRAMAÇÃO DO 8085.....	60
FIG. 3.3 – FUNCIONAMENTO DA PILHA NO 8085	63
FIG. 3.4 – MECANISMO DE CHAMADA DE UMA SUB-ROTINA	64
FIG. 5.1 – A MÁSCARA DE INTERRUPÇÕES VISTA PELA INSTRUÇÃO SIM	84
FIG. 5.2 – A MÁSCARA DE INTERRUPÇÕES VISTA PELA INSTRUÇÃO RIM.....	85
FIG. 6.1 – O MECANISMO DE SEGMENTAÇÃO	92
FIG. 6.2 – ENDEREÇAMENTO DE MEMÓRIA COM 8086.....	95
FIG. 6.3 – O MODELO DE PROGRAMAÇÃO DO 8086.....	95
FIG. 6.4 – EXEMPLO DE FORMATO DE INSTRUÇÃO DO 8086	98
FIG. 6.5 – DETALHAMENTO DO POSTBYTE	98
FIG. 6.6 – ENDEREÇAMENTO ABSOLUTO: MOV AX,[1000H]	100
FIG. 6.7 – ENDEREÇAMENTO INDIRETO: MOV AX,[BX]	101
FIG. 6.8 – ENDEREÇAMENTO INDEXADO: MOV AX,[100H+DI]	102

xxii	Microprocessadores: conceitos importantes
FIG. 6.9 – ENDEREÇAMENTO BASEADO: MOV [BX+0005H],CX	103
FIG. 6.10 – ENDEREÇAMENTO BASEADO INDEXADO: MOV AX,[BX+SI]	104
FIG. 7.1 – A PILHA COM DOIS PARÂMETROS.....	111
FIG. 7.2 – A PILHA APÓS O ARMAZENAMENTO DO VALOR DE BP	113
FIG. 7.3 – A PILHA EM UMA SUB-ROTINA FAR.....	113
FIG. 7.4 – A PILHA APÓS O RETORNO DA SUB-ROTINA	114
FIG. 7.5 – PASSAGEM DE PARÂMETROS EM SUB-ROTINAS ANINHADAS	116
FIG. 7.6 – CRIAÇÃO DE UMA VARIÁVEL LOCAL	117
FIG. 8.1 – A TABELA DE VETORES DE INTERRUPÇÃO DO 8086	122
FIG. 8.2 – A TABELA DE VETORES DE INTERRUPÇÃO DO 8086	123
FIG. 10.1 – O ABACUS	139
FIG. 10.2 – DETALHE DA JANELA PROGRAMA	141
FIG. 10.3 – A JANELA PROCESSADOR.....	143
FIG. 10.4 – A JANELA MEMÓRIA	144
FIG. 10.5 – A JANELA DISPLAY.....	145
FIG. 10.6 – A JANELA LEDS.....	145
FIG. 10.7 – A JANELA CHAVES	146
FIG. 10.8 – O ASSEMBLER DO ABACUS	146
FIG. 10.9 – A CAIXA DE DIÁLOGO SUB-ROTINAS.....	149
FIG. 10.10 – A CAIXA DE DIÁLOGO TRATADORES	149
FIG. 10.11 – DESVIOS DAS INSTRUÇÕES RST	150
FIG. 11.1 – O PROCESSO DE MONTAGEM (ASSEMBLY)	158
FIG. 11.2 – O PROCESSO DE LINKAGEM	159
FIG. 15.1 – CIRCUITO ELÉTRICO SIMULADO DA TECLA RST 7.5	181
FIG. 16.1 – REPRESENTAÇÃO DO COMPONENTE 8155	191
FIG. 16.2 – LIGAÇÃO DAS CHAVES E DOS LEDS AO 8155	191
FIG. 16.3 – O REGISTRADOR DE COMANDO DO 8155.....	193
FIG. A4.1 – ATRIBUTO DE VÍDEO EM MODO TEXTO.....	235

Lista de tabelas

TAB. 1.1 – A EVOLUÇÃO DA FAMÍLIA INTEL	34
TAB. 2.1 – OS DÍGITOS DO SISTEMA HEXADECIMAL E SEU VALOR.....	36
TAB. 2.2 – AS SOMAS ELEMENTARES NO SISTEMA HEXADECIMAL	37
TAB. 2.3 – A TABELA ASCII	46
TAB. 2.4 – A PÁGINA DE CÓDIGOS 850	47
TAB. 2.5 – OPERAÇÕES E MÁSCARAS PARA MANIPULAÇÃO DE BITS	49
TAB. 3.1 – SINAIS S0 E S1 DO 8085	59
TAB. 3.2 – SINAIS DO BARRAMENTO DE CONTROLE	60
TAB. 3.3 – FLAGS DO 8085	61
TAB. 4.1 – DIRETIVAS DA LINGUAGEM ASSEMBLY DO 8085.....	73
TAB. 5.1 – ENDEREÇOS DE DESVIO DAS INTERRUPÇÕES DO 8085	82
TAB. 5.2 – INTERRUPÇÕES DE HARDWARE E DE SOFTWARE.....	86
TAB. 6.1 – OS REGISTRADORES DE SEGMENTO DO 8086	94
TAB. 6.2 – SEGMENTOS DEFAULT E ALTERNATIVAS	107
TAB. 8.1 – INTERRUPÇÕES RESERVADAS DO 8086.....	126
TAB. 9.1 – AS INSTRUÇÕES DE MANIPULAÇÃO DE STRINGS	128
TAB. 9.2 – PREFIXOS DE REPETIÇÃO DO 8086	131
TAB. 12.1 – SUB-ROTINAS SIMULADAS EM ROM	163
TAB. 16.1 – ENDEREÇOS DAS PORTAS NO ABACUS	192
TAB. 17.1 – PRINCIPAIS DIRETIVAS DO ASM-86.....	195
TAB. A4.1 – VALORES DOS CAMPOS DO ATRIBUTO	235





Parte I – Teoria





1 Introdução aos computadores

1.1 Objetivo

Há algumas décadas, computadores vêm sendo empregados em escala crescente em instituições governamentais e militares, empresas, residências, bancos e instituições de ensino. O número de aplicações e de usuários é cada vez maior, pois a redução dos custos de produção dos microprocessadores tornou possível sua participação de nossa vida diária de forma natural e, muitas vezes, até despercebida. Estes componentes modificam nossa forma de comunicação com outras pessoas, a administração de nossos bens, nossa forma de trabalhar e nosso lazer.

Uma forma de acompanhar esta evolução consiste em observar as mudanças nas características dos computadores que vêm sendo produzidos. Algumas dessas características, em especial aquelas utilizadas na especificação de computadores pessoais, são hoje de conhecimento geral: o nome do microprocessador adotado, a capacidade da memória e do disco rígido, a quantidade de memória cache, a velocidade de acesso à Internet. Por isso, o leitor sabe que está diante de uma tecnologia que se renova dentro de um prazo muito curto.

Este fato se apresenta como um desafio especial para o ensino da disciplina de Microprocessadores. Pode-se pensar, a princípio, que esta tarefa consiste em manter os alunos a par dos últimos avanços dos grandes fabricantes e em atualizar, o mais rápido possível, o material didático. Uma reflexão mais profunda, porém, mostra que este não é o melhor caminho. O principal argumento contra este tipo de procedimento é a própria evolução da tecnologia: um aluno que recebesse, na metade do seu curso, conhecimentos específicos sobre o microprocessador mais atual, estaria desatualizado na época da sua formatura. Ensinar algo sobre microprocessadores não pode, portanto, significar apenas o estudo de um determinado componente. É preciso dar ao aluno condições para que ele mesmo seja capaz, hoje e depois de formado, de

descobrir e de entender os novos lançamentos da indústria de computadores. É preciso prepará-lo para decifrar os enigmas que o aguardam, ao invés de tentar prever todas as pedras em seu caminho e removê-las de antemão. Aqui não se pode dar o peixe, é preciso ensinar a pescar.

Por isso, o que se enfatiza neste livro são os conceitos que independem da evolução tecnológica e que, portanto, estão e continuarão a estar presentes em todos os microprocessadores, talvez com pequenas variações. É por isso, também, que é possível utilizar microprocessadores simples nos exemplos e exercícios da parte de laboratório. O que muda é o jeito de fazer cada coisa, mas as coisas feitas são essencialmente as mesmas. Desta forma, espera-se que o tempo e o esforço dedicados ao estudo das páginas seguintes sirvam não somente para um semestre, mas para toda uma vida.

1.2 Um pouco de história

A história dos computadores começou quando o homem percebeu que saber contar poderia ser útil para uma série de atividades essenciais à sua sobrevivência, como comparar quantidades de pessoas, controlar um estoque de alimentos ou saber quantos dias é preciso esperar para colher o que se plantou. Durante muitos anos, esses cálculos foram feitos com a utilização intuitiva dos dedos das mãos, ou de traços feitos em placas de barro úmido, ou ainda de pequenos objetos, como pedras ou pedaços de madeira.

Um marco importante na antigüidade foi a invenção do ábaco, creditada aos babilônios, que herdaram grande parte dos conhecimentos dos antigos sumérios, ocupantes da Mesopotâmia a partir do ano 5000 a. C. De lá, este instrumento se difundiu para o Egito, onde há registros de sua utilização em 500 a. C., e também para a China e a Índia, antes de se tornar conhecido na Europa. Ali, foi, durante séculos, o principal meio de fazer cálculos, pois o sistema de numeração romano, amplamente difundido, tornava difícil até mesmo uma simples adição (quanto é LXII + CXIV?). O ábaco foi caindo em desuso na Europa a partir do século XIII, quando a difusão do sistema de numeração indo-arábico facilitou a realização de cálculos manuais, mas ainda é utilizado em diversos países da Ásia.

Mesmo assim, permaneceu o método de cálculo mais eficiente até 1642, quando o matemático francês Blaise Pascal inventou uma máquina de calcular mecânica. Leibniz construiu uma versão mais avançada da máquina de Pascal um pouco mais tarde e, até o século XIX, surgiram vários outros projetos, grande parte dos quais não pôde ser implementada porque a indústria da época não tinha como produzir as peças com a precisão necessária. Devido à sua

construção, estas máquinas caracterizavam-se por uma grande rigidez em relação aos programas que podiam executar, de modo que seriam classificadas, atualmente, como máquinas dedicadas.

1.2.1 A primeira geração (1941 — 1955)

Já no século XX, os avanços da eletrônica, em particular o aperfeiçoamento da válvula, permitiram a construção de novos equipamentos para a realização de cálculos matemáticos – os computadores de *primeira geração*.

Os computadores modernos nasceram de um grande esforço feito em diferentes países durante a II Guerra Mundial. Em 1941, o inventor alemão Konrad Zuse produziu o Z3, utilizado no projeto de aviões e mísseis.

Nos Estados Unidos, o matemático Howard Aiken e os engenheiros Clair D. Lake, B.M. Durfee e F.E. Hamilton desenvolveram o Mark I, um computador eletromecânico que utilizava 3304 relés e ficou pronto em 1944. Sua principal utilização deu-se na criação de tabelas balísticas para melhorar a precisão da artilharia da marinha.

O primeiro computador eletrônico, Colossus, foi inventado pelo matemático Alan Turing, na Inglaterra, e ficou pronto em 1943. Foi projetado e utilizado especificamente para decifrar códigos de comunicação do exército alemão e sua existência foi mantida em segredo durante décadas.

Contrastando com a aplicação dedicada do Colossus, o primeiro computador eletrônico de propósito geral foi o ENIAC (Electronic Numerical Integrator and Calculator). Foi projetado pelos engenheiros John W. Mauchly e J. Presper Eckert, Jr., da Universidade da Pensilvânia, nos Estados Unidos, e entrou em operação em 1946. Ocupava uma área de 140 m², pesava 30 toneladas e utilizava cerca de 18000 válvulas, que consumiam 200 kW e eram interligadas por 800 km de fios. O calor gerado era imenso, exigindo um sistema de ventilação forçada. Havia também uma equipe de técnicos, que ficava continuamente substituindo as válvulas que se queimavam durante a operação.

Embora as válvulas funcionassem apenas como chaves, os computadores construídos até esta época não utilizavam o sistema de numeração binário para representar dados. Levados pela tradição de contar no sistema decimal, os projetistas agrupavam as válvulas em arranjos que representavam números decimais. Além disso, as instruções a executar não tinham a forma de programas armazenados na memória, mas eram determinadas por um conjunto

de chaves e conexões em painéis de comando. A modificação destes conceitos coube a um matemático chamado John von Neumann.

1.2.2 A máquina de von Neumann (1949)

John Louis von Neumann nasceu em 1903, na Hungria. Suas habilidades matemáticas extraordinárias já eram evidentes aos seis anos, quando era capaz de dividir, mentalmente, números de até oito algarismos. Estudou nas universidades de Budapeste e de Berlim e na Escola Técnica Superior de Zurique, formando-se em Engenharia Química. Obteve em seguida o título de doutor em Matemática pela Universidade de Budapeste e, a partir de 1930, passou a trabalhar em Princeton, EUA. Ali, foi escolhido para ser um dos seis professores de Matemática do Instituto de Estudos Avançados, quando da sua fundação, em 1933. As atividades desse instituto incluíam temas relacionados à solução de problemas computacionais – o próprio Alan Turing estudou ali de 1936 a 1938.

Dentro deste ambiente, von Neumann se interessou pelo projeto de computadores e decidiu criar o seu próprio. Introduziu na indústria o conceito de programa armazenado, ao compreender que seria possível representar programas sob forma numérica e, portanto, armazená-los em memória da mesma maneira que os dados. Percebeu ainda que a aritmética decimal poderia ser substituída, com vantagens, pela aritmética binária. Criou, assim, um modelo para representar sua máquina, que é, ainda hoje, a base da arquitetura de quase todos os computadores.

A figura 1.1 ilustra esse modelo, que compreende cinco componentes: a memória, a unidade aritmética e lógica (ALU), a unidade de controle e as unidades de entrada e de saída. As instruções que a máquina era capaz de executar permitiam transferir dados entre esses componentes e realizar operações matemáticas elementares com eles.

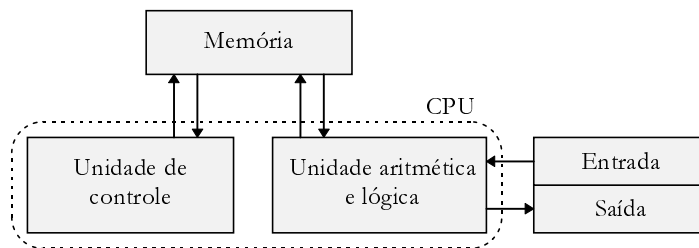


Fig. 1.1 – Modelo da máquina de von Neumann

O conjunto da unidade de controle e da ALU ficou também conhecido como *unidade central de processamento*, ou CPU, do inglês *central processing unit*. Convém lembrar que a divisão do modelo nesses componentes é uma abstração que não se reflete diretamente na construção real do computador. Nos primeiros computadores, em particular, cada bloco ocupava muito espaço e utilizava centenas e até milhares de componentes discretos.

1.2.3 A segunda geração (1955 – 1965)

Com a invenção do transistor, em 1948, o mundo dos computadores foi tomado de assalto por uma onda de novos projetos que deu origem, na década de 60, a duas empresas que se tornaram mundialmente conhecidas: DEC e IBM. Os computadores PDP-1, PDP-8, IBM-7090, IBM-7094, CDC-6000, IBM-7030 e Gamma-60 foram alguns dos produtos que surgiram neste período. No entanto, os elevados custos dessas máquinas restringiam sua viabilidade a aplicações estratégicas do governo e a grandes empresas.

1.2.4 A terceira geração (1965 – 1980)

No final da década de 50, surgiram os primeiros circuitos integrados. Arranjos de algumas dezenas de transistores, como os utilizados para fazer flip-flops ou portas lógicas, foram substituídos por componentes únicos, que ocupavam menos espaço e eram mais fáceis de montar. Os custos caíram, criando uma nova faixa de mercado, que incluía empresas de médio porte, universidades e centros de pesquisa. As características físicas dos novos componentes reduziram os tempos de propagação dos sinais elétricos e proporcionaram, com isso, um aumento de desempenho. Juntas, estas mudanças caracterizaram uma nova geração de máquinas, da qual são exemplos o IBM-360 e os minicomputadores da série PDP-11, da DEC.

Até a geração anterior, a execução de um programa bloqueava a CPU, mesmo que esta estivesse apenas aguardando a resposta de algum equipamento periférico. A fim de aproveitar esses intervalos de ociosidade, surgiu a idéia da *multiprogramação*, que consiste em instalar diversos programas simultaneamente na memória do computador. Isso não torna possível a execução simultânea, pois a CPU é única e somente um desses programas pode estar em execução num determinado instante. No entanto, os programas podem ser executados ciclicamente, em pequenos intervalos de tempo, excluindo-se os que não precisam de atenção. Esta técnica recebe, em inglês, o nome de *time sharing*. Se

a velocidade de execução é suficientemente alta, os usuários dos diferentes programas têm a impressão de que todos são executados simultaneamente.

1.2.5 A quarta geração (1980 – ...)

A partir dos anos 80, o grande desenvolvimento da tecnologia de circuitos integrados fez com que o número de transistores que podiam ser integrados numa pastilha de silício atingisse a faixa dos milhares e, logo em seguida, dos milhões. Isto se refletiu em nova queda de preços e novo aumento de capacidade dos computadores.

Surgiram então os computadores pessoais, que passaram a ser utilizados para processamento de texto, cálculos auxiliados por planilhas eletrônicas e em projetos gráficos, atividades para as quais os grandes computadores não eram bem adaptados.

A interligação dos computadores pessoais, primeiramente através de redes locais e, logo depois, através da Internet, aliada ao emprego da multimídia, multiplicou muitas vezes as possibilidades de aplicação dessas máquinas.

Além disso, a redução de custo e de volume dos componentes produzidos permitiu sua aplicação nos assim chamados sistemas embutidos, que controlam aeronaves, embarcações, automóveis e equipamentos de pequeno porte.

A capacidade de cálculo das máquinas atuais supera em diversas ordens de magnitude aquela dos instrumentos primitivos. Embora seja teoricamente possível fazer com um ábaco os mesmos cálculos que se fazem com os microprocessadores, a capacidade e a velocidade destes tornaram possíveis aplicações que antes eram inimagináveis.

Mesmo assim, a comparação entre ábacos e microprocessadores mostra que o estado atual da tecnologia, em particular da informática, é uma manifestação de um desejo permanente do homem de descobrir verdades e de prever fatos através do cálculo e, com isso, de ampliar seu domínio sobre a natureza.

1.3 A família Intel

A Intel, fundada em 1968, produzia memórias de computador, até que a Busicom, fabricante de calculadoras do Japão, lançou o desafio de se construir uma unidade central de processamento num único circuito integrado. A Intel conseguiu atender o pedido, anunciando o novo componente em 1971. Por tratar os dados em grupos de 4 bits, este recebeu o nome de 4004. Pouco depois, foi lançada uma versão de oito bits, denominada 8008. Estas foram as

primeiras CPU's integradas num único chip – *os primeiros microprocessadores*. A produção iniciou em pequena escala, pois a empresa não imaginava o interesse que esses componentes despertariam. Sem saber, estava iniciando uma era marcada por uma evolução surpreendente.

Em função do sucesso alcançado, a Intel passou a projetar um novo chip para ultrapassar a barreira dos 16 kB de memória, limite imposto pelo número de pinos do 8008. Em 1974, lançou o 8080, que foi amplamente utilizado, inclusive com aplicações em aparelhos domésticos. Dois anos mais tarde, foi lançado o 8085, uma versão do 8080 com modificações nos sistemas de interrupção e de entrada / saída.

Em seguida, apareceram a primeira CPU de 16 bits, o 8086, e sua versão com barramento de 8 bits, o 8088. Esta variante do 8086 mostrou-se interessante para facilitar a migração de produtos que usavam o 8085 para a nova arquitetura de 16 bits. O 8088 tornou-se um padrão em termos de computador pessoal, porque foi escolhido pela IBM para compor o PC-XT.

Os microprocessadores 80186 e 80188 são extensões do 8086 e do 8088, respectivamente. Incluem na própria pastilha alguns periféricos, como controladores de DMA e temporizadores, que os tornam interessantes para uso dedicado como microcontroladores. Entretanto, nunca foram chips altamente utilizados.

A evolução dos processadores até este ponto motivou a produção de software mais sofisticado. Isto trouxe um aumento vertiginoso da demanda de espaço de endereçamento, que sentimos até os dias atuais, apesar do enorme avanço já conseguido. Ao mesmo tempo, a complexidade dos programas aumentava, e tornou-se claro que havia necessidade de melhorar sua organização. Era importante impedir que um programa usuário tivesse acesso a recursos críticos para o funcionamento global do computador, que deveriam ser administrados apenas pelo sistema operacional. Com estes fatores em mente, a Intel lançou o 80286, capaz de endereçar diretamente até 16 MB de memória quando opera no assim chamado *modo protegido*, o qual oferece também o suporte necessário à atribuição de direitos de acesso aos diferentes programas em execução. O processador suporta, ainda, o modo de endereçamento virtual, que permite estender o espaço de endereçamento a 1 GB. No assim chamado *modo real*, o 80286 comporta-se como o 8086.

O 80286 foi utilizado no PC-AT e repetiu o sucesso do 8088. O passo seguinte e natural foi a passagem para 32 bits, concretizada com o lançamento do 80386. Este foi seguido rapidamente pelo 80486, o primeiro a possuir um coprocessador matemático on-chip. Depois disso, foi lançada a família Pentium, cada vez com mais refinamentos para aumentar o desempenho.

No período descrito, passou-se de uma CPU de 4 bits a uma de 32 bits, com desempenho mais de 1000 vezes superior. A tabela 1.1 apresenta algumas características que ilustram essa evolução.

Nome	Ano	Dados	Endereços*	Comentário
4004	1971	4 bits	1 kB	primeiro microprocessador
8008	1972	8 bits	16 kB	primeiro micro 8 bits
8080	1974	8 bits	64 kB	supera barreira dos 16 kB; amplo uso
8085	1976	8 bits	64 kB	8080 com E/S modificada
8086	1978	16 bits	1 MB	primeira CPU 16 bits num chip
8088	1980	16 bits	1 MB	8086 com barram. 8 bits; IBM-PC
80186	1982	16 bits	1 MB	8086 + periféricos on-chip
80188	1982	16 bits	1 MB	8088 + periféricos on-chip
80286	1982	16 bits	16 MB (1GB)	proteção e endereçamento virtual
80386	1985	32 bits	4 GB (70 TB)	primeira CPU 32 bits
80386SX	1988	32 bits	4 GB (70 TB)	80386 com barramento 16 bits
80486	1989	32 bits	4 GB (70 TB)	mais rápido; coprocessador on-chip
Pentium	1993-...	32 bits	4 GB (70 TB)	diversos modelos, evoluindo

Tab. 1.1 – A evolução da família Intel

Mais informações podem ser obtidas na Internet, a partir do item [Links Interessantes](#), na homepage que dá suporte a este livro.

* Os valores entre parênteses são alcançados utilizando endereçamento virtual.

2 Conceitos básicos

2.1 Sistemas de numeração

Os sistemas de numeração que nos interessam são os sistemas ditos posicionais, aqueles em que a contribuição de cada algarismo, ou dígito, para o valor da quantidade representada depende da posição que ele ocupa no número. O sistema decimal, que usamos no dia-a-dia, é posicional. Por exemplo, a contribuição do dígito “1” é de uma unidade no número 51 e de dez unidades no número 17.

Contrastam com esse tipo de sistema os sistemas não posicionais, tais como os números romanos, em que a contribuição do dígito “I” é de uma unidade em I e em cada uma das posições de II. Os sistemas não posicionais são pouco práticos para a realização de cálculos e por isso não serão abordados em maior detalhe.

Existem dois conceitos fundamentais para a compreensão do texto que segue e que, embora não representem propriamente uma novidade, merecem ser explicitados: é preciso distinguir claramente entre o *valor* de um número e sua *representação*.

O *valor* de um número corresponde à quantidade que ele representa, ao passo que a *representação* desse número corresponde aos dígitos que escrevemos para simbolizá-lo. No sistema decimal, por exemplo, representamos a quantidade doze (valor) pelo número “12” (representação). No sistema hexadecimal, que veremos logo a seguir, a mesma quantidade é representada pelo número “C”.

Ao longo do presente texto, utilizaremos o sistema decimal sempre que for necessário fazer referência ao valor de um número. Haverá, então, uma coincidência entre o valor 12 e sua representação no sistema decimal (12), por causa da necessidade de se escolher algum sistema para falar sobre o valor dos números. Para evitar confusão, é importante estar atento para saber quando o texto se refere ao valor de um número, que nunca muda, e quando se refere à sua representação, que depende do sistema escolhido.

Um sistema posicional é especificado em termos de uma constante denominada *base*, que determina a relação entre o valor de um número e sua representação através da expressão:

$$Valor = \sum_{i=0}^{n-1} d_i B^i, \quad (2.1)$$

em que:

- d_i é o i -ésimo dígito do número, contado da direita para a esquerda,
- n é o número de dígitos e
- B é a base.

Por exemplo, no sistema decimal, que recebe este nome por ter base 10, o valor do número 1234 é dado por $4 * 10^0 + 3 * 10^1 + 2 * 10^2 + 1 * 10^3$, que corresponde à quantidade 1234. No sistema binário, o valor de 1101 é dado por $1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3$, que corresponde à quantidade 13.

É importante notar também que, quando usamos sistemas de bases diferentes, muda o número de símbolos de que precisamos para representar seus dígitos. Um sistema de numeração de base N utiliza N dígitos diferentes. Isto significa que, para sistemas de base menor do que 10, nem todos os dígitos de 0 a 9 serão utilizados. Por exemplo, o sistema octal (base 8) utiliza apenas os dígitos de 0 a 7, e o sistema binário apenas os dígitos 0 e 1. Por outro lado, sistemas de base maior do que 10 precisam de símbolos novos para representar os dígitos além do 9. O sistema hexadecimal, que será largamente utilizado ao longo do texto, precisa de símbolos para representar as quantidades de 0 a 15. Os dígitos de 0 a 9 foram mantidos para as quantidades de 0 a 9, como no sistema decimal, e as letras de A até F escolhidas para as quantidades de 10 a 15. A tabela 2.1 mostra os dígitos do sistema hexadecimal e seu valor.

Dígito	0	...	9	A	B	C	D	E	F
Valor	0	...	9	10	11	12	13	14	15

Tab. 2.1 – Os dígitos do sistema hexadecimal e seu valor

Voltando à equação 2.1, o valor do número hexadecimal 4D2 é dado por $2 * 16^0 + D * 16^1 + 4 * 16^2$, que corresponde à quantidade 1234. Dizemos então que 1234 é dado por 4D2 em hexadecimal.

É também importante estabelecer uma notação para especificar a base em que se representa um valor, para evitar confusão. Sempre que necessário, números binários serão seguidos pela letra Y, números octais pela letra Q, números decimais pela letra T e números hexadecimais pela letra H. Isto permitirá

distinguir números como 100Y de 100H. Números sem sufixo devem ser considerados decimais.

A habilidade de fazer cálculos simples no sistema hexadecimal é muito valiosa quando se trabalha com programação de microprocessadores, da mesma forma como a habilidade de fazer cálculos simples no sistema decimal é útil em nossas atividades diárias. Por isso, convém estender a capacidade aprendida de somar desde $0 + 0$ até $9 + 9$, de forma a incluir também os novos dígitos de A até F. A tabela 2.2 apresenta as somas novas no sistema hexadecimal.

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Tab. 2.2 – As somas elementares no sistema hexadecimal

2.2 Conversão de base

A conversão de base é um processo que permite obter a representação de um valor em um dado sistema de numeração a partir de uma representação conhecida em outro sistema, de base diferente.

Na seção anterior, vimos que 1234T é equivalente a 4D2H, porque calculamos os valores de ambas e estes coincidiram. Mas, em geral, não teremos a sorte de adivinhar a representação desejada, e por isso precisamos de um algoritmo para passar de um sistema a outro. O algoritmo dado a seguir resolve este problema e pode ser implementado em um programa simples, conforme feito no capítulo 14. A tarefa é determinar a representação da quantidade N na base B .

1. Divida N por B ; sejam q_0 e r_0 o quociente e o resto dessa operação, respectivamente; seja ainda $i = 0$;
2. se q_i for igual a zero, a representação de N na base B é $r_i r_{i-1} r_{i-2} \dots r_1 r_0$ e a conversão está terminada. Caso contrário, incremente i e continue;
3. divida q_{i-1} por B ; sejam q_i e r_i o quociente e o resto dessa divisão, respectivamente. Volte ao passo 2.

A figura 2.1 mostra as divisões efetuadas para obter a representação do valor 1234 no sistema hexadecimal.

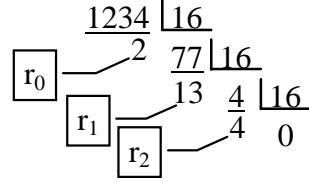


Fig. 2.1 – O algoritmo de conversão de base

Dividimos 1234 por 16, que dá $q_0 = 77$ e $r_0 = 2$. Continuando, dividimos 77 por 16 e obtemos $q_1 = 4$ e $r_1 = 13$ (ou DH) e então dividimos 4 por 16, que dá $q_2 = 0$ e $r_2 = 4$. Neste ponto, paramos e alinhamos os restos na forma $r_2r_1r_0$ para obter o resultado esperado, 4D2H.

A seguir, analisaremos a fundamentação matemática do algoritmo dado. Começamos com o caso trivial em que o número a converter é tal que o algoritmo pára logo após a primeira divisão, quando então temos:

$$N = q_0B + r_0 = 0 \cdot B + r_0 = r_0, \quad (2.2)$$

o que permite escrever N como r_0 .

Se forem necessárias duas divisões, teremos:

$$N = q_0B + r_0 \quad (2.3)$$

e

$$q_0 = q_1B + r_1 = 0 \cdot B + r_1 = r_1. \quad (2.4)$$

Substituindo (2.4) em (2.3), vem:

$$N = r_1B + r_0, \quad (2.5)$$

o que permite escrever N como r_1r_0 .

Para o caso de três divisões, teremos:

$$N = q_0B + r_0, \quad (2.6)$$

$$q_0 = q_1B + r_1 \quad (2.7)$$

e

$$q_1 = q_2B + r_2 = 0 \cdot B + r_2 = r_2 \quad (2.8)$$

Substituindo (2.8) em (2.7) e depois em (2.6), temos:

$$N = r_2B^2 + r_1B + r_0, \quad (2.9)$$

que permite escrever N como $r_2r_1r_0$.

O algoritmo é baseado na generalização destes resultados para números arbitrariamente grandes.

2.2.1 Casos particulares de conversão de base

Quando a conversão que se deseja fazer deve acontecer entre bases tais que uma é potência inteira da outra, pode-se utilizar um atalho.

Sejam B_{orig} e B_{dest} as bases de origem e de destino, respectivamente. No caso em que a base de destino é maior do que a de origem e existe um número inteiro n tal que:

$$B_{dest} = B_{orig}^n, \quad (2.10)$$

é possível agrupar os dígitos da representação de origem de n em n e atribuir, a cada grupo formado, um dígito da representação de destino.

Como exemplo, seja a tarefa de converter para o sistema hexadecimal o número binário 1101 1000 0111 0110. Como a base de destino (16) é potência inteira da base de origem (2) porque ambas estão relacionadas pela igualdade $16 = 2^4$, o valor de n é 4. Isso significa que podemos agrupar os dígitos do número binário de 4 em 4 e escrever, para cada grupo, um dígito hexadecimal. De fato, $1101 = D$, $1000 = 8$, $0111 = 7$ e $0110 = 6$, de modo que a representação procurada é D876H.

Inversamente, no caso em que a base de destino é menor do que a de origem e existe um número inteiro n tal que:

$$B_{orig} = B_{dest}^n, \quad (2.11)$$

é possível desagrupar os dígitos da representação de origem em n dígitos da representação de destino.

Por exemplo, seja a tarefa de converter o número octal 14273 para binário. Como a base de origem (8) é potência inteira da base de destino (2) porque ambas estão relacionadas pela igualdade $8 = 2^3$, o valor de n é 3. Isso significa que podemos desagrupar os dígitos do número octal em grupos de 3 dígitos binários. De fato, $1 = 001$, $4 = 100$, $2 = 010$, $7 = 111$ e $3 = 011$, de modo que a representação procurada é 001 100 010 111 011.

Finalmente, é possível combinar os dois métodos e fazer conversões entre bases que, embora não sejam potência inteira uma da outra, são potências de um mesmo número menor. É o caso das bases 8 e 16, que não são potências uma da outra, mas são ambas potências de 2.

Como exemplo, seja a tarefa de converter o número octal 14273 para hexadecimal. Convertamos primeiramente 14273 para o sistema binário, obtendo 001 100 010 111 011. Em seguida, rearranjamos os dígitos binários em grupos de 4, da direita para a esquerda, completando o último com zeros à esquerda, se necessário. Obtemos então 0001 1000 1011 1011 que, convertido para hexadecimal, resulta em 18BBH.

2.3 Representação de números inteiros

Ao se iniciar o estudo dos microprocessadores, é importante deixar claro como se faz a representação das quantidades numéricas envolvidas, merecendo especial atenção os números inteiros com e sem sinal. [Mors88] e [WS99] trazem informações que complementam o material aqui exposto.

Para representar números positivos, utiliza-se normalmente o valor do próprio número binário. Assim, por exemplo, as quantidades 5 e 7 são representadas por 0101 e 0111, respectivamente.

Para os números negativos, a convenção escolhida é a que representa os números em complemento de 2, descrita com mais detalhes na seção 2.3.1.

Num computador, sempre existe um limite para o maior número que se pode representar, dado que a memória, por maior que seja, é finita. Este limite varia de um computador para outro e, para os exemplos que seguem, suporemos, sem perda de generalidade, que os registradores disponíveis para representar nossos números são de 4 bits. Então podemos representar ao todo $2^4 = 16$ números diferentes, de 0000 a 1111. Se tentássemos continuar além do limite 1111, teríamos $1111 + 1 = 10000$. Mas, como poderíamos representar apenas os quatro dígitos menos significativos, estaríamos de volta ao 0000, pois 10000 tem, por hipótese, um bit a mais do que o permitido.

Esta volta à origem sugere dispor os números ao redor de um círculo, e não ao longo de um eixo infinito, como se faz na matemática convencional. Esta representação aparece na figura 2.2, que permite visualizar o que acontece quando fazemos operações de soma e subtração. Para executar a soma de dois números a e b , basta encontrar a representação de a no círculo e avançar b posições no sentido horário. Para fazer a subtração $a - b$, por outro lado, basta recuar b posições a partir de a , no sentido anti-horário. Exemplos de cálculos possíveis são $5 + 3$ ou $6 - 4$.

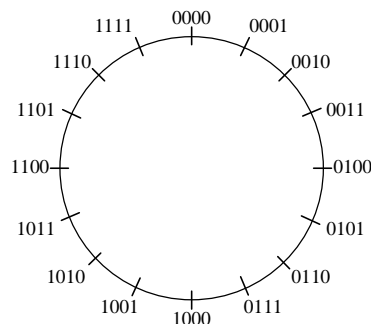


Fig. 2.2 – Representação circular dos números de 4 bits

A figura 2.2 também mostra o que acontece quando o resultado de uma soma excede o maior número representável, no caso, 1111. Se tentarmos somar $10 + 6$, por exemplo, chegamos a 0000 e não a 10000, que seria o resultado correto, exatamente como acontece quando somamos dois números num processador e não temos como registrar o “vai 1”.

O modelo apresentado até aqui é suficiente para os casos em que não aparecem valores negativos nos cálculos realizados. No entanto, sabemos que existem grandezas que podem assumir valores negativos, e por isso é natural perguntar o que acontece quando o resultado de uma operação é um número negativo. Se, na figura 2.2, tentarmos representar o cálculo de $0 - 1$, começamos na posição 0000 e recuamos uma posição no sentido anti-horário, o que nos leva a 1111. Da mesma forma, os números negativos subsequentes ficariam em 1110, 1101, 1100, etc. Poderíamos continuar desta forma e chegar até o número 0001, que representaria então o valor -15. Mas aí ficaríamos sem números positivos, e por isso é necessário definir uma fronteira que separe os números positivos dos negativos.

Costuma-se considerar positivos os números cujo bit mais significativo é 0 e negativos os números cujo bit mais significativo é 1, o que divide ao meio o conjunto de números representáveis. Isto também facilita a tarefa de determinar o sinal de um número em um programa, pois basta analisar esse bit. No caso dos números de 4 bits, ficamos com 8 números negativos e 8 números positivos, considerando-se o zero incluído nestes últimos. A disposição desses números ao redor do círculo aparece na figura 2.3.

Note que, agora, a capacidade de representação de números não vai mais de 0 a 15, mas sim de -8 até +7. Para representar números além destes limites, seria preciso adotar registradores maiores, por exemplo de 8, 16 ou 32 bits.

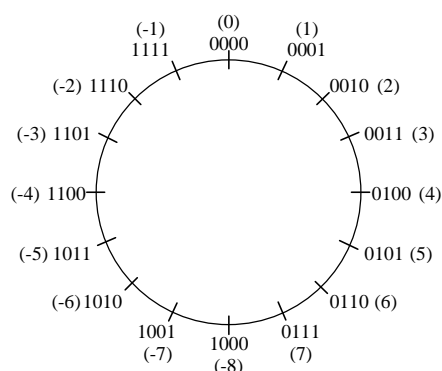


Fig. 2.3 – Representação de números positivos e negativos

No caso geral de números de n bits, o aspecto do círculo é semelhante ao da figura 2.3. O número de divisões é sempre igual a 2^n , os números positivos vão de 0 até $2^{n-1} - 1$ e os números negativos de -1 até -2^{n-1} . Estes números são encontrados na documentação de compiladores de linguagens de alto nível, como C ou Pascal, quando se apresentam os limites da capacidade de representação dos diferentes tipos de variáveis inteiras.

2.3.1 Obtenção do valor simétrico de um número

Escolhida a representação para os números positivos e negativos, passaremos a investigar se existe uma maneira fácil de, dado um número a , obter seu simétrico ($-a$). Se compararmos diretamente as representações de 1 e -1, 2 e -2, e assim por diante, conforme representado pelas linhas tracejadas da figura 2.4, não veremos, de imediato, qualquer coisa que pareça facilitar a obtenção de uma a partir da outra.

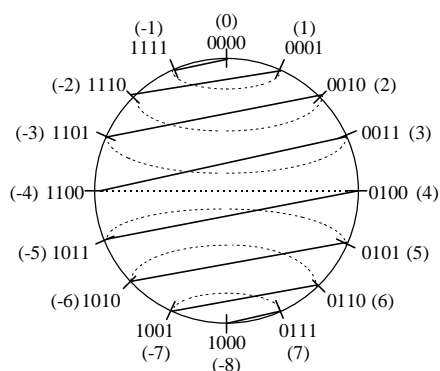


Fig. 2.4 – Relações de complemento entre simétricos

No entanto, se compararmos as representações de 1 e -2, e 2 e -3, e assim por diante, conforme representado pelas linhas cheias, veremos que uma pode ser obtida da outra trocando-se os zeros por 1's e vice-versa. O resultado obtido com esta operação chama-se *complemento de 1* do número em questão, e por isso podemos dizer que o simétrico de um número a do círculo pode ser obtido através da expressão:

$$-a = (\text{complemento de 1 de } a) + 1. \quad (2.12)$$

O membro direito dessa expressão é conhecido como *complemento de 2* de a . Pode-se mostrar (como?) que

$$(\text{complemento de 2 de } a) = 2^n - a, \quad (2.13)$$

em que n é o número de bits utilizados para representar os números.

Note ainda que a expressão 2.12 funciona independentemente de ser a um número positivo ou negativo. Por exemplo, para obter a representação de -3 partindo de 0011 (3), complementamos os bits (1100) e somamos 1, obtendo 1101. Inversamente, para obter a representação de 5 a partir de 1011 (-5), complementamos (0100) e somamos 1 (0101 = 5).

2.3.2 Subtração usando adição

A representação em complemento de 2 apresenta ainda uma outra vantagem, que é a de permitir a realização de subtrações utilizando a adição. Isto é particularmente interessante na implementação de circuitos digitais, uma vez que o mesmo circuito pode ser utilizado para executar ambas as operações.

O ponto de partida para fazer a subtração $a - b$ com auxílio da adição é observar que, se podemos representar números de, no máximo, n bits, é verdade que

$$a - b = a + 2^n - b, \quad (2.14)$$

pois, uma vez que caminhamos sobre o círculo, o fator 2^n representa uma volta completa e não afeta o resultado. Pela equação 2.13,

$$a - b = a + (\text{complemento de 2 de } b). \quad (2.15)$$

Por exemplo, o resultado de $6 - 4$ pode ser obtido somando-se 0110 (6) ao complemento de 2 de 0100 (4), que é 1100 (-4), desconsiderando-se o “vai 1”, que não tem como ser registrado: $0110 + 1100 = (1)0010$.

Outra forma de visualizar o funcionamento deste artifício consiste em observar, na figura 2.4, que somar 1100 a 0110 é o mesmo que dar uma volta completa no círculo e voltar 4 posições. Por isso, quando se parte de 6, chega-se a 2, que é o resultado correto.

2.4 A representação BCD

Uma tarefa que se apresenta com frequência na elaboração de programas é a apresentação de um resultado numérico, contido num registrador do processador, em algum tipo de mostrador, tal como um display de cristal líquido. Nestes casos, o programador normalmente tem à sua disposição uma sub-rotina que apresenta o valor hexadecimal contido no registrador diretamente no mostrador. Se o valor que está contido no registrador, que é sempre hexadecimal, deve ser mostrado no sistema decimal, então é necessário primeiramente obter a representação decimal desse valor, para depois

apresentá-lo. Esta conversão de base, que não deixa de ser trabalhosa, pode ser evitada quando se trabalha com números no formato BCD (*binary coded decimal*), decimal codificado como binário.

O formato BCD utiliza o sistema hexadecimal. No entanto, são permitidos apenas os dígitos de 0 a 9, o que permite ler os números como decimais. Por exemplo, o número 12T (decimal) é representado como 12H, e não como 0CH, que seria o usual. Desta forma, se for necessário apresentar esse valor num mostrador, o resultado será correto.

Esta abordagem tem, naturalmente, a limitação de que a representação utilizada para o número 12 representa, na verdade, a quantidade 18 ($2 + 1 * 16$), e isso causa problemas quando se fazem operações com esses números. No entanto, é possível elaborar regras simples que permitem fazer adições e subtrações com números BCD. Para tanto, colocamos inicialmente os dígitos do sistema hexadecimal no círculo da figura 2.5.

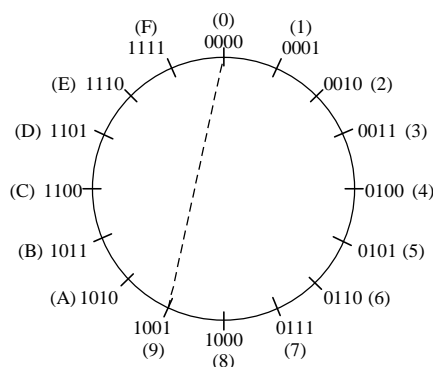


Fig. 2.5 – Os dígitos do sistema hexadecimal e um atalho

Observamos então que a soma de dois dígitos será correta enquanto não ultrapassar 9. Por exemplo, $2H + 6H = 8H$ é o mesmo que $2T + 6T = 8T$. Quando o valor passa de 9, porém, podem surgir letras no resultado (como em $2 + 8 = AH$) ou então o resultado pode estar incorreto (como em $9 + 9 = 12H$). Em qualquer dos casos, o problema é que as seis letras de A a F deveriam ser “saltadas” para que se obtivesse o resultado desejado, como se, ao invés de avançar sobre o círculo, fosse utilizado o atalho representado pela linha tracejada da figura 2.5. Isto permite elaborar o seguinte algoritmo para a soma de números BCD:

- somar os dígitos como hexadecimais;
- se a soma for superior a 9, somar mais 6.

Exemplo: na soma $84 + 26$, ambos números BCD, deveríamos obter 110, também BCD (suponha, para este exemplo, que temos como registrar números

de 3 dígitos). Mas a soma hexadecimal simples resulta em AAH, resultado não válido como BCD. Aplicando o algoritmo, temos os seguintes passos:

- somamos 4 com 6; como o resultado (AH) é superior a 9, somamos mais 6 e obtemos 10H; escrevemos o 0 como dígito menos significativo do resultado e “vai 1”;
- somamos esse 1 com 8 e com 2 e obtemos BH, também superior a 9. Então somamos novamente mais 6 e obtemos 11H que, colocado à esquerda do 0 obtido anteriormente, dá 110H, que é a representação correta do número BCD desejado. A figura 2.6 ilustra a operação.

$$\begin{array}{r} 1 \\ 84 \\ + 26 \\ \hline BA \\ + 66 \\ \hline 110 \end{array}$$

Fig. 2.6 – A soma de números BCD

A subtração de números BCD é análoga. A figura 2.5 permite concluir que, nesse caso, o problema aparece quando é preciso “emprestar 1” e que a correção consiste em subtrair mais 6 sempre que isso acontecer, o que equivale a percorrer o atalho tracejado no sentido inverso. Por exemplo, num processador de 8 bits, o cálculo $45 - 37$, na ausência de correção, dá o resultado hexadecimal 0EH. Subtraindo mais 6, obtém-se o resultado BCD desejado, 8.

A maioria dos microprocessadores tem uma instrução que implementa a correção para a soma de dois números BCD, mas não para a subtração. Por isso, surge aqui uma aplicação importante da subtração através da soma, empregando o complemento de 10. O complemento de 10 de um número é obtido acrescentando-se 1 ao seu complemento de 9. Este, por sua vez, é o número cujos dígitos representam a quantidade que falta aos dígitos correspondentes no número original para chegar a 9. Por exemplo, o complemento de 9 de 1234 é 8765, e o complemento de 9 de 23 é 76. Os complementos de 10 desses números são, então, 8766 e 77, respectivamente. Note que, a exemplo do que acontece com os números binários e o complemento de 2, o complemento de 10 é tal que a soma de um número com seu complemento é 10^N , onde N é a quantidade de dígitos do número em questão: $1234 + 8766 = 10000$ e $23 + 77 = 100$.

Por isso, a subtração $a - b$ pode ser realizada como $a + (10^N - b)$, onde N é o número de dígitos disponíveis para representar os valores a e b . Haverá um “vai 1”, que excede o número de bits da representação e não será registrado.

A subtração BCD discutida acima, 45 - 37, seria feita assim:

- obter o complemento de 10 de 37, que é $62 + 1 = 63$;
- somar 45 com 63, que dá A8H;
- corrigir o resultado pelo algoritmo *da soma* BCD, obtendo 108H;
- descartar a centena, uma vez que, por hipótese, a representação era de 8 bits, obtendo o resultado 08.

2.5 Tabelas de codificação de caracteres

Os dispositivos de entrada e saída dos computadores precisam, com frequência, trocar informações sob forma de texto, o que envolve caracteres alfanuméricos.

Cada um desses caracteres tem que ser codificado por um número binário e, para que seja possível trocar com facilidade informações entre diferentes computadores, é conveniente que essa codificação seja normalizada. Com este intuito, surgiu, na década de 60, a tabela ASCII (*American Standard Code for Information Interchange*), que atribuiu os números de 00H a 7FH aos 128 caracteres considerados mais usuais na época. Note que isto significa que os caracteres ASCII precisam de 7 bits para sua codificação.

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	☺	☹	♥	♦	♣	♠	Bip	Bk	Tab	LF	♂	♀	CR	🎵	⚙
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	▶	◀	↕	!!	¶	§	-	↑	↑	↓	→	←	L	↔	▲	▼
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	Spc	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Tab. 2.3 – A tabela ASCII

Com o passar do tempo, surgiu a necessidade de padronizar a representação de caracteres acentuados, caracteres utilizados em molduras de janelas de texto e muitos outros. Passou-se então estender a tabela ASCII para 8 bits. Contudo, constatou-se que, para satisfazer os usuários de todos os países que utilizam

computadores, seriam necessários muito mais do que os 256 caracteres que se podem representar dessa forma. Por isso, surgiram várias extensões, denominadas páginas de código (*code pages*). Nessas tabelas, os primeiros 128 caracteres são idênticos aos da tabela ASCII e os demais variam de acordo com as necessidades da língua do usuário. No Brasil, a página de códigos mais utilizada é página de códigos 850, apresentada na tabela 2.4.

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	😊	😄	♥	♠	♣	♠	Bip	Bk	Tab	LF	♂	♀	CR	🎵	⚙
1	▶	◀	↕	!!	¶	§	—	↑	↑	↓	→	←	L	↔	▲	▼
2	Spc	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
8	Ç	ü	é	â	ã	ä	å	ç	ê	ë	è	ï	î	í	Ä	Å
9	É	æ	Æ	ô	õ	ö	ù	û	ÿ	Ö	Ü	ø	£	Ø	×	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	@	¬	½	¼	¿	«	»
B	☐	☐	☐		†	Á	Â	Ã	©	¶		¶	¶	¢	¥	ℓ
C	L	⌞	⌞	⌞	—	†	ã	Â	ℓ	¶	¶	¶	¶	=	¶	□
D	ð	Ð	Ê	Ë	È	⌞	Í	Î	Ï	⌞	⌞	⌞	⌞	⌞	⌞	⌞
E	Ó	ß	Ô	Õ	ö	Ö	µ	Þ	þ	Ú	Û	Ü	Ý	Ý	—	ˆ
F	.	±	=	¾	¶	§	÷	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	■	

Tab. 2.4 – A página de códigos 850

Existem ainda muitas outras extensões, sobre as quais é possível obter informações detalhadas na Internet. As tabelas, acompanhadas de um bom histórico de todo o esforço de normalização, bem como perguntas e respostas freqüentes sobre o assunto podem ser encontradas a partir do item [Links Interessantes](#), na homepage que dá suporte ao livro.

Como a programação de microprocessadores envolve freqüentemente o trabalho com caracteres alfanuméricos, é conveniente saber de cor que:

- letras maiúsculas iniciam em 41H;
- letras minúsculas iniciam em 61H e diferem das maiúsculas apenas por 1 bit;
- os dígitos de 0 a 9 ocupam a faixa 30H a 39H.

2.6 Manipulação de bits

A manipulação de bits individuais dentro de um byte é de grande importância no desenvolvimento de programas para microprocessadores. Aparece freqüentemente na manipulação de caracteres alfanuméricos ou na interação com hardware externo através de portas paralelas de entrada e saída. Com a manipulação de bits é fácil transformar letras minúsculas em maiúsculas e vice-versa ou determinar se um número é par ou ímpar. No caso de uma porta de entrada, pode ser necessário determinar o valor de um de seus bits, a fim de determinar a posição de uma chave ligada externamente ao pino que corresponde a esse bit. No caso de uma porta de saída, pode ser necessário setar, zerar ou complementar um certo bit sem perturbar o estado dos demais, para evitar que os dispositivos conectados a estes sejam ligados ou desligados indevidamente.

A manipulação de bits é possível graças à presença das assim chamadas *operações lógicas* no conjunto de instruções dos microprocessadores, em geral AND, OR e XOR. Estas instruções aceitam dois bytes como operandos e resultam em um byte cujos bits são obtidos através da aplicação dessas operações aos bits dos operandos, tomados aos pares. Por exemplo, 45H AND 73H, respectivamente 0100 0101 e 0111 0011, resulta em 0100 0001 ou 41H. Da mesma forma, 45H OR 73H = 77H e 45H XOR 73H = 36H. Confira!

As instruções apresentadas permitem alterar o valor de bits individuais dentro de um byte, desde que se escolham convenientemente a operação e o segundo operando a ser utilizado, que é conhecido por *máscara*.

As aplicações mais comuns de manipulação de bits são as seguintes:

- *setar um ou mais bits de um byte*: a operação lógica utilizada é OR e a máscara escolhida de modo que os bits correspondentes às posições que devem ser setadas estejam em 1 e os demais em 0. Por exemplo, para setar os bits 1 e 3 do byte 1101 0100 sem alterar os demais, escolhemos como máscara 0000 1010. O resultado da operação OR com esses argumentos é 1101 1110;

- *zerar um ou mais bits de um byte*: a operação lógica utilizada é AND e a máscara escolhida de modo que os bits correspondentes às posições que devem ser zeradas estejam em 0 e os demais em 1. Por exemplo, para zerar os bits 1 e 3 do byte 1101 1110 sem alterar os demais, escolhemos como máscara 1111 0101. O resultado da operação AND com esses argumentos é 1101 0100;
- *complementar um ou mais bits de um byte*: a operação lógica utilizada é XOR e a máscara escolhida de modo que os bits correspondentes às posições que devem ser complementadas estejam em 1 e os demais em 0. Por exemplo, para complementar os bits 0 e 7 do byte 1101 1110 sem alterar os demais, escolhemos como máscara 1000 0001. O resultado da operação XOR com esses argumentos é 0101 1111;
- *testar se um ou mais bits de um byte estão setados*: a operação lógica utilizada é AND e a máscara escolhida de modo que os bits correspondentes à posição que deve ser testada estejam em 1 e os demais em 0. Por exemplo, para testar se o bit 3 de um byte qualquer está setado, escolhemos como máscara 0000 1000. O resultado da operação AND com esses argumentos será igual a zero se o bit 3 não estiver setado, e diferente de zero (igual a 08H) se o bit 3 estiver setado. Estes dois resultados possíveis (igual a zero, diferente de zero) podem ser explorados mediante emprego das instruções de desvio condicional dos microprocessadores, que permitem fazer com que trechos de código distintos sejam executados para cada caso.

A tabela 2.5 resume as operações e máscaras para os tipos de manipulação apresentados.

Objetivo	Máscara	Operação
Setar	1 para setar; 0 nos demais	OR
Zerar	0 para zerar; 1 nos demais	AND
Complementar	1 para complementar; 0 nos demais	XOR
Testar	1 para testar; 0 nos demais	AND

Tab. 2.5 – Operações e máscaras para manipulação de bits

2.7 Tipos de memórias

A indústria eletrônica, num constante esforço para aumentar a capacidade de armazenamento dos chips de memória, já empregou diversas técnicas de produção, que resultaram em vários tipos de memórias. Do ponto de vista da

disciplina de Microprocessadores, porém, o que interessa é distinguir as memórias cujo conteúdo pode ser alterado pelo programa (como as memórias RAM, SRAM e DRAM) daquelas cujo conteúdo permanece fixo (como as ROM's e EPROM's). Por isso, o texto se refere às memórias fixas em geral como ROM's e, às voláteis, como RAM's.

Projetos de computadores precisam distinguir as regiões da memória onde um programa precisa escrever daquelas onde é suficiente ter acesso apenas para ler. É preciso também separar os conteúdos de memória que não podem ser perdidos quando o sistema é desligado daqueles que sempre podem ser reconstruídos pelo programa e que podem, portanto, residir em memória volátil. Finalmente, é preciso levar em conta que o acesso à memória RAM é mais rápido do que o acesso à ROM. Trabalhando com estas restrições, o projetista determina onde utilizar cada tipo de memória. Em geral, a região a partir da qual o processador busca a primeira instrução a executar assim que recebe alimentação é mapeada em ROM. É o caso, por exemplo, do BIOS (*Basic Input-Output System*) dos PC's, que reside em ROM na placa-mãe e é responsável por carregar o sistema operacional do disco rígido para outras regiões da memória, mapeadas em RAM.

Em sistemas menores, dedicados, é comum que o programa resida completamente em ROM, de modo que não há necessidade de se carregá-lo a partir de um disco. Mesmo nesse tipo de sistema, porém, o uso de RAM é quase sempre indispensável, pois é comum que o programa executado necessite de uma área para guardar resultados temporários.

2.8 Operações destrutivas e não destrutivas

As operações de leitura de uma palavra da memória ou de um registrador de um processador são não destrutivas, ao passo que as operações de escrita são destrutivas. Estes termos significam que o conteúdo de uma palavra da memória ou de um registrador permanece inalterado quando é lido e que, ao se sobrescrever esse conteúdo com outro valor, o valor original é perdido.

É útil ter estas propriedades em mente quando se começa a estudar o funcionamento de programas para microprocessadores. É por isso que se pode afirmar, por exemplo, que uma instrução de movimentação de dados, tal como MOV A,B, a qual copia para o registrador A o conteúdo do registrador B (e não o contrário!), destrói o conteúdo anterior do registrador A, mas preserva o conteúdo de B. Se, em um dado instante, A contém o valor 10 e B contém 20, então a execução da instrução acima deixa ambos os registradores com 20.

2.9 Interpretação do conteúdo da memória

A execução de um programa por um processador envolve constantemente o acesso à memória, seja para a leitura das instruções que devem ser executadas, seja para o acesso aos dados lá armazenados. Isto significa que a memória armazena instruções executáveis e também números inteiros, reais e caracteres alfanuméricos.

Todos esses conteúdos diferentes são representados por seqüências de um ou mais números binários. A interpretação desses números é feita pelo processador e pelo programa, na medida em que o valor em questão é lido como uma instrução, como um número ou como um caracter alfanumérico. Por exemplo, dado que uma posição de memória contém o valor 47H, isto pode ser interpretado das seguintes formas:

- se este valor aparecer em um programa para o microprocessador 8085, fará com que este execute a instrução MOV B,A;
- como número inteiro, representará a quantidade 71 (47H);
- como caracter alfanumérico, corresponderá à letra G, de acordo com a tabela ASCII.

O estudo dos microprocessadores envolve freqüentemente situações em que é necessário analisar o conteúdo de uma certa região de memória. É preciso ter em mente, portanto, que a interpretação desse conteúdo não pode ser feita simplesmente olhando-se para os valores, mas que é necessário também saber como programa em questão fará uso deles.

2.10 A execução de programas

O microprocessador é o elemento encarregado da execução dos programas armazenados na memória de um computador. Corresponde à unidade central de processamento (CPU) e, no contexto do modelo da figura 1.1, compreende a unidade de controle, responsável pela leitura e pela decodificação das instruções e a unidade lógica e aritmética (ALU), que efetua operações como a adição e as operações lógicas AND e OR.

A CPU contém, ainda, uma área de memória de alta velocidade, dividida em *registradores*, utilizados no processamento das instruções. O número desses registradores e as operações que podem ser feitas com cada um deles varia de uma CPU para outra, embora existam alguns que são comuns a todos os microprocessadores, por serem essenciais para o seu funcionamento.

Pelo menos um desses registradores precisa ter funcionalidade para executar operações aritméticas e lógicas e recebe, em geral, o nome de *acumulador (A)*. Outro registrador importante é o contador de programa (*program counter, PC* às vezes também chamado de *instruction pointer, IP*), onde o processador armazena o endereço de memória da próxima instrução a ser executada. A execução de uma instrução atualiza automaticamente este registrador, de modo que ele está sempre apontando para a instrução seguinte, que é lida assim que termina a execução da instrução em andamento. Um terceiro exemplo é o apontador da pilha (*stack pointer, SP*), utilizado para controlar o acesso a uma área de memória utilizada para armazenar valores temporários, como parâmetros passados para sub-rotinas e endereços de retorno de sub-rotinas e de tratadores de interrupção. A figura 2.7 ilustra os elementos citados.

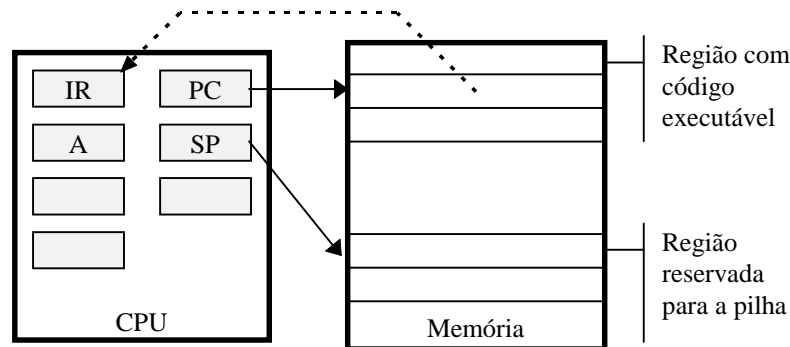


Fig. 2.7 – Alguns elementos da CPU e da memória

Existem ainda registradores não acessíveis ao programador, que a CPU utiliza internamente. É o caso do registrador de instrução (*instruction register, IR*), onde são armazenadas as instruções lidas da memória, para que possam ser decodificadas e executadas.

2.10.1 A execução de uma instrução

A execução de uma instrução pela CPU pode ser descrita, de forma simplificada, pelos seguintes passos:

- *leitura da instrução a executar*: os códigos hexadecimais que compõem a instrução são lidos a partir do byte endereçado pelo contador de programa (PC) e copiados para o registrador de instrução (IR);
- *atualização do contador de programa*: o número de bytes lidos no passo anterior é somado ao PC, que passa então a apontar para a instrução seguinte;

- *decodificação da instrução*: a CPU identifica a unidades responsável pela execução da instrução (por exemplo, a ALU, se for uma operação aritmética) e se serão há necessidade de dados externos (por exemplo, valores contidos na memória ou de uma porta de entrada);
- *leitura dos dados, se necessário, para os registros internos da CPU*: dados externos são copiados para registradores internos, onde podem ser manipulados;
- *execução da instrução*: a instrução é levadas a efeito (obtem-se o resultado de uma soma, por exemplo);
- *armazenamento de resultados*: eventuais resultados são transferidos para seus lugares de destino;
- *fim do ciclo*: a CPU retorna ao passo 1 para executar a instrução seguinte.

Esta seqüência é conhecida como *ciclo de leitura-decodificação-execução* e representa o princípio de funcionamento de todo microprocessador.

2.11 A memória e os barramentos

A memória armazena os programas em execução e os dados processados por estes programas. Consiste de um conjunto de células, denominadas *palavras* ou *bytes*, que são identificadas univocamente por um número, denominado *endereço físico* ou simplesmente *endereço*. Pode-se imaginar uma memória de m bytes como uma tabela de m linhas, em que os endereços são usados para numerar as linhas e cada linha tem capacidade para conter um byte.

Cada byte é composto de um número fixo de dígitos binários ou *bits*. Desta forma, um byte composto de n bits pode assumir 2^n valores diferentes, de 0 a $2^n - 1$. Os bytes da máquina de von Neumann, por exemplo, tinham 40 bits, o que permitia a representação de duas instruções de 20 bits ou então de um número inteiro de 39 bits com um bit de sinal. Atualmente, é convenção estabelecida que $n = 8$, e por isso o valor que cada byte pode assumir varia de 0 a 255. Diz-se que o valor assumido por um byte é o *dado* armazenado no endereço correspondente.

Há situações em que os 256 valores distintos que um byte pode assumir não são suficientes para representar uma grandeza com que se trabalha. Nestes casos, agrupam-se dois ou mais bytes para formar arranjos de mais bits, o que aumenta exponencialmente a capacidade de representação. Diz-se que dois bytes (16 bits) formam um *word*, quatro bytes (32 bits) formam um *doubleword* e oito bytes (64 bits) formam um *quadword*.

O número n de bits de um byte tem influência sobre o número de linhas físicas através das quais acontece a transferência de dados entre o processador e a

memória. O conjunto dessas linhas físicas é denominado *barramento de dados*, e o número de linhas que o constitui é a sua *largura*. A largura do barramento de dados de um processador é, em geral, um múltiplo de n . São comuns barramentos de dados com larguras de 8, 16 e 32 bits.

Através do barramento de dados é possível tanto ler quanto modificar o conteúdo da memória, e o processador precisa de um mecanismo que lhe permita indicar qual dessas operações deve ser efetuada. Para tanto, existe o *barramento de controle*. Os diversos barramentos que interligam o processador à memória aparecem na figura 2.8.

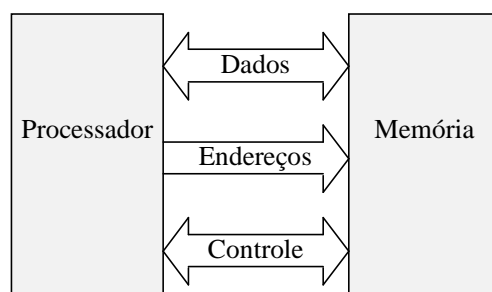


Fig. 2.8 – Conexão de um microprocessador à memória

O barramento de controle contém, tipicamente, os sinais RD (read) e WR (write), que distinguem as duas formas de acesso, além de um sinal que diz se este acesso se refere, de fato, à memória ou a outro dispositivo.

O processador precisa também de um mecanismo para especificar o endereço do byte que pretende acessar. Este processo recebe o nome de *endereçamento* de um byte e se dá através do *barramento de endereços*, que é outro conjunto de linhas físicas que conectam o processador à memória, independente do barramento de dados.

A fim de endereçar um determinado byte, o processador coloca a combinação de bits que corresponde ao seu endereço, expresso no sistema de numeração binário, nas linhas desse barramento. Desta forma, o número máximo de bytes que podem ser endereçados por um processador é igual a 2^m , em que m é a largura do barramento de endereços. São comuns barramentos de 16 e 32 bits, que correspondem, respectivamente, a capacidades de endereçamento de 64 kB (2^{16}) e 4 GB (2^{32}). Os endereços ocupam a faixa de 0 a $2^m - 1$.

A memória da máquina de von Neumann, por exemplo, tinha capacidade para $2^{12} = 4096$ palavras de 40 bits, numeradas de 0 a 4095. As instruções, que, conforme mencionado, tinham 20 bits, eram compostas de dois campos: um campo de 8 bits para especificar o tipo de instrução e outro, de 12 bits, para endereçar uma das 4096 palavras da memória.

A utilização de dígitos binários tem um inconveniente: não é prático trabalhar com números de 8, 16 ou 32 algarismos. Por isso, adota-se uma representação alternativa para esses números, utilizando o sistema hexadecimal em lugar do sistema binário. A passagem de um sistema para outro é simples, pois os dígitos hexadecimais correspondentes a um número binário podem ser obtidos agrupando-se os bits de 4 em 4, conforme exposto na seção 2.2.1. Desta forma, o número de algarismos fica dividido por 4.

A figura 2.9 ilustra uma memória de bytes de 8 bits, endereçados através de endereços de 16 bits.

Endereço (16 bits)	Dado (8 bits)
0000H	35H
0001H	04H
0002H	30H
0003H	01H
0004H	19H
0005H	71H
0006H	01H
0007H	06H
0008H	19H
0009H	62H
000AH	FFH
000BH	E1H
...	...
FFFEH	ABH
FFFFH	CDH

Fig. 2.9 – Memória de 64 kB

Neste caso, existem 2^{16} endereços diferentes, que permitem endereçar ao todo $2^{16} = 65536$ bytes, ou ainda 64 kB (convenciona-se que $1\text{kB} = 2^{10} \text{ bytes} = 1024 \text{ bytes}$). Os endereços ocupam a faixa que vai de 0 a $2^{16}-1 = 65535 = \text{FFFFH}$.

Os valores dos dados foram escolhidos ao acaso dentro do intervalo que vai de 00H a 255 = FFH. Os três barramentos apresentados trabalham em conjunto para realizar cada acesso do processador à memória, seja para ler um byte de uma instrução de programa, seja para ler ou modificar algum dado.

Quando executa uma instrução que exige a leitura da memória, o processador coloca primeiramente o endereço do byte a ser lido no barramento de

endereços e indica, através do barramento de controle, que se trata de uma operação de leitura. A memória reage, então, colocando no barramento de dados o valor do byte endereçado, que o processador copia para algum registrador.

Uma operação de escrita também inicia com a ativação do barramento de endereços, mas o barramento de controle informa à memória que se trata de uma operação de escrita. Esta reage, então, copiando o dado que o processador coloca no barramento de dados para a célula endereçada.

3 Introdução ao 8085

3.1 Origem e características técnicas

O microprocessador 8085 foi lançado em 1976, como sucessor do 8080. Este tinha apenas uma entrada de interrupção e não tinha sistema de comunicação serial, o que dificultava seu uso em projetos considerados interessantes pela Intel. O gerador de clock e o controlador de sistema eram externos, mas o ideal seria poder integrá-los com a CPU. Além disso, havia necessidade de duas tensões de alimentação (+5V e -5V), quando o ideal seria usar apenas uma.

O novo processador satisfaz todos estes requisitos. A fim de manter a compatibilidade de software com o 8080, o conjunto de instruções deste foi mantido. Acrescentaram-se apenas as instruções RIM e SIM, necessárias por causa das novas interrupções e do sistema de comunicação serial. A velocidade de execução aumentou em 50%, devido à mudança da frequência de clock de 2,6 para 3,0 MHz e a outras modificações internas.

O microprocessador 8085 é encapsulado em um circuito integrado de 40 pinos, com os sinais externos distribuídos conforme indicado na figura 3.1.

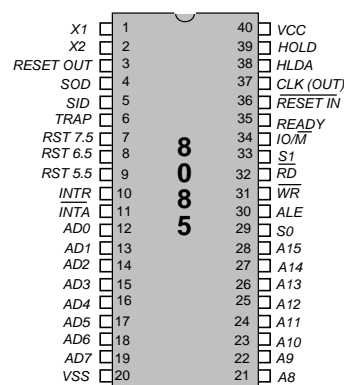


Fig. 3.1 – Distribuição dos sinais no circuito integrado 8085

Os pinos Vcc (+5V) e Vss, respectivamente pinos 40 e 20, recebem a tensão de alimentação. Os sinais X1 e X2 (pinos 1 e 2) correspondem às entradas do gerador de clock interno. A estas entradas conecta-se um circuito oscilador (cristal, rede RC, etc.) para a geração do clock. CLK (pino 37) é uma saída na qual se tem o sinal de clock do 8085, que pode ser utilizado por circuitos periféricos que compõem um sistema. A frequência deste sinal é igual à metade da frequência do sinal gerado em X1 e X2.

O barramento de endereços do 8085 é composto de 16 linhas, o que lhe confere uma capacidade de endereçamento de 64 kB. Os sinais A8 a A15 (pinos 21 a 28) correspondem aos 8 bits mais significativos do barramento de endereços; AD0 a AD7 (pinos 12 a 19) são linhas utilizadas tanto para endereços quanto para dados, o que constitui uma novidade com relação ao 8080, que tinha linhas separadas para dados e endereços. O duplo uso destas linhas foi introduzido no 8085 para acomodar todos os seus sinais no mesmo número de pinos do 8080, a fim de que fosse possível utilizar o mesmo tipo de pastilha para os dois microprocessadores, e tornou-se comum a partir desta época. A multiplexação de dados e endereços é realizada em função dos ciclos de barramento do microprocessador. No primeiro ciclo de uma instrução de acesso à memória, estes pinos correspondem aos oito bits menos significativos do barramento de endereços. Nos demais ciclos, correspondem aos oito bits do barramento de dados. O sinal ALE (Address Latch Enable, pino 30) permite ao hardware externo distinguir os dados dos endereços e é concebido para ser usado como sinal de clock para um latch que captura os endereços.

Os pinos SOD e SID (4 e 5) fazem a saída e a entrada de dados de forma serial, respectivamente. Estas operações são realizadas através das instruções SIM (Set Interrupt Mask) e RIM (Read Interrupt Mask) que, além de habilitar ou inibir interrupções, permitem enviar ou receber dados nesses pinos.

TRAP, RST7.5, RST6.5, RST5.5 e INTR (pinos 6 a 10) correspondem a entradas que permitem sinalizar pedidos de interrupção de hardware. INTA (pino 11) é um sinal de reconhecimento de interrupção, enviado em resposta a um pedido de interrupção feito pela entrada INTR.

HOLD (pino 39) é uma entrada que permite a um dispositivo externo requisitar o direito de utilização dos barramentos de dados e de endereços. Quando esta entrada é ativada, o 8085 entra em estado de alta impedância, permitindo que um dispositivo externo assuma o controle dos barramentos. Isto permite a implementação do acesso direto à memória (DMA).

A saída HLDA (Hold Acknowledge, pino 38) indica ao dispositivo externo que requisitou o controle dos barramentos que estes estão acessíveis.

READY (pino 35) é uma entrada que oferece suporte ao mecanismo de geração de estados de espera (wait states) para as operações de leitura em memória ou E/S. Após ter requisitado a leitura de um dado, o 8085 se coloca em estado de espera até que o dispositivo ative essa entrada, indicando que o dado está disponível no barramento.

RESET IN (pino 36) permite reinicializar o 8085. O contador de programa assume o valor 0000H, fazendo com que a execução do programa recomece a partir deste endereço. Esta operação também desabilita as interrupções, mas não modifica o conteúdo dos demais registradores, da máscara de interrupções ou dos flags. A saída RESET OUT (pino 3) é utilizada para sinalizar a reinicialização do processador a outros circuitos do sistema, que podem então reagir de forma conveniente.

S0 e S1 (29 e 33) são saídas que indicam o estado do 8085, de acordo com a tabela 3.1. São úteis em testes de sistemas em desenvolvimento.

S0	S1	Estado
0	0	Retenção
0	1	Escrita
1	0	Leitura
1	1	Interrupção

Tab. 3.1 – Sinais S0 e S1 do 8085

Os sinais IO/M, RD e WR (pinos 34, 32 e 31, respectivamente) formam o barramento de controle. Conforme apresentado na seção 2.11, RD e WR dizem se a operação é de leitura ou de escrita, respectivamente.

Para entender o papel do sinal IO/M, é preciso saber que os barramentos de dados e de endereços não são conectados somente à memória, mas também compartilhados com outros dispositivos, formando o espaço de endereços de entrada e saída (IO). Assim, quando o processador coloca um endereço no barramento, este é percebido tanto pela memória quanto por esses dispositivos periféricos. Por isso, existem instruções de acesso à memória e instruções de acesso ao espaço de E/S. A diferença entre elas está na ativação (ou não) do sinal IO/M, que é utilizado pelos circuitos de decodificação de endereços para determinar se devem selecionar a memória ou um dispositivo periférico.

A tabela 3.2 apresenta as combinações possíveis e um exemplo de instrução para cada uma delas. Os sinais RD e WR nunca são ativados simultaneamente. Os traços na tabela indicam estado de alta impedância (3-state).

IO/M	WR	RD	Significado	Exemplo
0	0	-	Escrita em memória	STA 2100H
0	-	0	Leitura em memória	LDA 2100H
1	0	-	Escrita em E/S	OUT 80H
1	-	0	Leitura em E/S	IN 80H
-	-	-	Operação interna	INR A

Tab. 3.2 – Sinais do barramento de controle

3.2 O modelo de programação

O modelo de programação de um microprocessador é uma abstração utilizada no desenvolvimento de programas. É independente da arquitetura do hardware e sua função é prover o programador de uma representação que facilite o entendimento e a memorização do conjunto de instruções. Inclui os registradores e os flags do processador e, dependendo da complexidade deste, podem aparecer também dispositivos periféricos, como portas de entrada e saída.

O modelo também é útil para avaliar a capacidade de processamento de um microprocessador, pois o tamanho e a quantidade dos registradores permitem estimar sua capacidade de endereçamento e, com a prática, saber o que esperar do seu conjunto de instruções. É um bom hábito iniciar o estudo de um processador novo por este modelo.

O modelo de programação do 8085 é apresentado na figura 3.2.

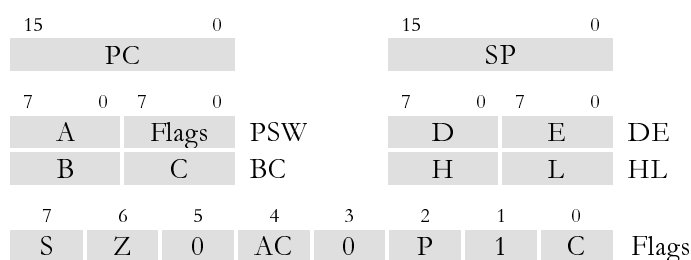


Fig. 3.2 – Modelo de programação do 8085

Os registradores PC e SP, ambos de 16 bits, são o *Program Counter* e o *Stack Pointer*. Desempenham as funções delineadas na seção 2.10.

B, C, D, E, H e L são registradores de 8 bits, de uso geral. Aparecem aos pares no modelo de programação porque algumas instruções os afetam em conjunto, como se formassem registradores de 16 bits.

Um exemplo de instrução que afeta um par de registradores é INX H, que incrementa o par HL, da seguinte forma:

- soma 1 ao valor contido em L;
- se houver “vai 1”, soma 1 ao valor contido em H.

O acumulador (A) participa da grande maioria das operações lógicas e aritméticas. As instruções que realizam operações de dois argumentos, tais como a soma, usam como primeiro argumento sempre o acumulador. Por isso, os mnemônicos dessas instruções especificam apenas o segundo argumento, que pode ser uma constante ou outro registrador. É o caso da instrução ADD B, que acrescenta o valor do registrador B ao acumulador.

O acumulador faz parte de uma palavra de 16 bits denominada PSW (*program status word*), em cujo byte menos significativo residem os flags. Estes são variáveis binárias, modificadas individualmente pelo resultado de diversas operações, em especial as lógicas e aritméticas. Note que há apenas 5 flags. Os bits não utilizados têm valores fixos e não são importantes. A tabela 3.3 relaciona os flags do 8085 e os critérios que determinam seu valor.

FLAG	Condição necessária e suficiente para flag = 1
Z (Zero)	Resultado de uma operação é nulo
S (Sign)	Resultado de uma operação é negativo
P (Parity)	Resultado de uma operação tem paridade par
C (Carry)	Operação exige “vai 1” ou “empresta 1”
AC (Aux. C)	Operação exige “vai 1” do bit 3 para o bit 4

Tab. 3.3 – Flags do 8085

É importante salientar que, embora os flags sejam afetados pelos resultados das operações executadas, eles não refletem necessariamente o conteúdo do acumulador. Por exemplo, a operação SUB A zeraria o acumulador e deixaria este flag em 1. Se, logo em seguida, fosse executada a instrução MVI A,01H, o conteúdo do acumulador passaria a ser diferente de zero, mas o flag permaneceria setado, porque a instrução MVI não afeta os flags. Portanto, o simples fato de o *zero flag* estar em 1 não garante que o acumulador esteja zerado.

Outro caso importante é o das instruções de comparação. Toda comparação efetua, internamente, uma subtração, sem no entanto alterar o conteúdo do acumulador. Por exemplo, se o acumulador contém o número 35H e é

executada a instrução CPI 35H, a subtração feita internamente colocará o *zero flag* em 1 para sinalizar que o resultado da comparação é uma igualdade. O valor do acumulador, porém, continua sendo 35H. Os exemplos apresentados ilustram a importância de se observar as regras segundo as quais os flags são posicionados quando se escreve um programa. As tabelas do anexo 1 dão informações sobre como as instruções do 8085 afetam os flags.

3.3 A pilha

A pilha (ou *stack*, em inglês) é uma região da memória utilizada pelo processador para armazenar valores temporários, como parâmetros para sub-rotinas e endereços de retorno. É um conjunto de bytes onde se pode escrever e ler, respeitando as seguintes regras:

- a pilha deve ser vista tal qual uma pilha de pratos de cozinha. Bytes novos são colocados no topo da pilha, e a remoção dos bytes tem que ser feita na ordem inversa da sua colocação. Não vale puxar um prato do meio da pilha: em qualquer instante, apenas o último byte empilhado está acessível;
- a pilha é controlada pelo registrador SP (*ponteiro da pilha ou stack pointer*), que aponta para o último elemento empilhado; é dever do programa inicializar este registrador com a instrução LXI SP,<endereço>, para que a área escolhida seja corretamente endereçada;
- a transferência de dados entre pilha e CPU envolve sempre um par de bytes (16 bits). Na escrita, o byte mais significativo é movido primeiro, e depois o menos significativo. Na leitura, a ordem é a inversa;
- estas operações só são permitidas para os seguintes registradores:
 - ◊ PSW (Program Status Word, contendo acumulador e flags);
 - ◊ os pares de registradores BC, DE e HL;
- as instruções utilizadas na movimentação são PUSH (escrita) e POP (leitura), seguidas do argumento que identifica os registradores envolvidos. PUSH PSW, por exemplo, coloca o PSW na pilha;
- quando uma instrução faz referência a um par de registradores, indica somente o nome do registrador mais significativo do par. Por exemplo, para transferir os dois bytes do topo da pilha para o par de registradores BC, escreve-se apenas POP B, e não POP BC.

A figura 3.3 ilustra o funcionamento da pilha no caso da execução da sequência de instruções

```
PUSH B
POP D
```

numa situação em que os registradores B e C contêm, respectivamente, os valores 01H e 02H. As operações realizadas copiam estes valores para o par DE.

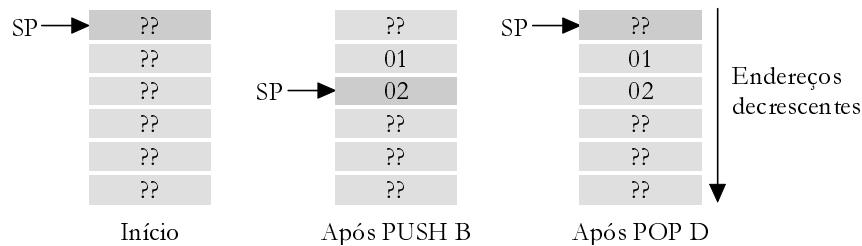


Fig. 3.3 – Funcionamento da pilha no 8085

No início, o conteúdo da pilha pode ser qualquer e é ignorado pelo programa. A execução de uma instrução PUSH consiste dos seguintes passos:

1. o valor de SP é decrementado de 1;
2. o conteúdo do registrador mais significativo do par (no caso do exemplo, o registrador B) é copiado para o endereço contido em SP;
3. o valor de SP é decrementado de 1;
4. o conteúdo do registrador menos significativo do par (no caso do exemplo, o registrador C) é copiado para o endereço contido em SP; este passa a ser o novo topo da pilha. Esta é a situação intermediária na figura acima.

A execução de uma instrução POP consiste dos seguintes passos:

1. o byte endereçado por SP é copiado para o registrador menos significativo do par especificado (no caso do exemplo, o registrador E);
2. o valor de SP é incrementado de 1;
3. o byte endereçado por SP é copiado para o registrador mais significativo do par especificado (no caso do exemplo, o registrador D);
4. o valor de SP é incrementado de 1; este passa a ser o novo topo da pilha.

As regras apresentadas têm algumas implicações que vale a pena ressaltar:

- a leitura da pilha não altera o conteúdo da área de memória utilizada;
- o fato de os valores serem retirados da pilha sempre na ordem inversa da sua colocação não deve gerar a falsa idéia de que os valores precisam ser lidos para dentro dos mesmos registradores de onde vieram. A pilha pode, de fato, ser utilizada para trocar o conteúdo de diferentes pares de registradores, como no fragmento de código abaixo, que troca o conteúdo do par BC com o do par DE:

```
PUSH B
PUSH D
POP B
POP D
```

- a pilha cresce no sentido dos endereços decrescentes;
- é responsabilidade do programador cuidar para que a pilha não cresça a ponto de invadir áreas de memória destinadas para outros fins (como por exemplo uma região de dados ou o próprio código do programa).

3.4 Sub-rotinas

Sub-rotinas são trechos de código para os quais o processamento pode ser desviado mediante instruções de controle de fluxo. Executado o código correspondente, um mecanismo de retorno faz com que a execução do programa continue na instrução seguinte à que provocou o desvio. Costuma-se denominar esse desvio de *chamada* da sub-rotina e a volta ao ponto de continuação de *retorno*, conforme a figura 3.4.

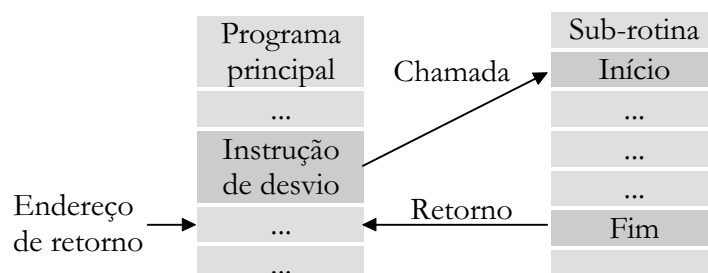


Fig. 3.4 – Mecanismo de chamada de uma sub-rotina

São duas as razões que podem tornar interessante a divisão de um programa em sub-rotinas:

- podem abrigar trechos de código que executam funções específicas, o que permite organizar melhor o programa e torná-lo mais facilmente compreensível por terceiros;
- podem ser chamadas de qualquer lugar do programa, tantas vezes quantas se desejar. Assim, quando uma tarefa precisa ser executada repetidas vezes, o código correspondente só precisa ser escrito uma única vez.

3.4.1 O papel da pilha na chamada de sub-rotinas

A principal instrução do 8085 utilizada para chamar uma sub-rotina é a instrução `CALL <endereço de destino>`. Esta instrução provoca um desvio incondicional para o endereço fornecido, que deve ser o endereço de início da sub-rotina; existem ainda outras, como `CC` (call if carry), `CZ` (call if zero), etc.,

que executarão o desvio somente se a condição correspondente estiver satisfeita. Em qualquer dos casos, quando o desvio acontece, o processador passa a ler as instruções a partir do endereço fornecido. Contudo, deve continuar no endereço da instrução seguinte ao call (*endereço de retorno*) após o término da sub-rotina chamada. Isto exige o armazenamento do endereço de retorno, feito mediante uso da pilha. O processamento das instruções do tipo CALL consiste dos seguintes passos:

- inicia pelo incremento do PC, que passa a apontar para o endereço da instrução seguinte à que foi lida. Este é, por definição, o endereço de retorno. O processador copia este valor do PC para a pilha, com a correspondente diminuição do valor de SP de duas unidades. Assim, o topo da pilha passa a conter o endereço de retorno;
- o endereço de destino fornecido na instrução de chamada é copiado para o PC, fazendo com que a próxima instrução executada seja a primeira instrução da sub-rotina chamada.

O processamento de uma instrução CALL é, portanto, equivalente a um PUSH do PC (embora não exista a instrução PUSH PC), seguido de um JMP para o endereço de destino.

O término de uma sub-rotina é indicado pela instrução RET (*return*), processada como segue:

- os dois bytes do topo da pilha são removidos e colocados no PC (program counter), de maneira que a próxima instrução a ser executada será aquela endereçada por estes dois bytes. Considerando que o topo da pilha contenha o endereço de retorno salvo pela instrução CALL, o programa continuará sua execução conforme desejado.

O processamento de uma instrução RET é, portanto, equivalente a um POP para o PC (embora não exista a instrução POP PC).

É fundamental compreender que o retorno ao ponto de execução correto só será possível se, no instante da execução da instrução RET, o topo da pilha contiver o endereço de retorno. Se isto não for verdade, seja porque a sub-rotina deixou SP com um valor incorreto, seja porque o endereço de retorno foi adulterado, o programa será desviado para um endereço sem sentido e muito provavelmente se perderá. Isto não quer dizer que a sub-rotina não tenha direito de utilizar a pilha – pelo contrário, é até comum que ela chame suas próprias sub-rotinas –, mas sim que toda sub-rotina tem a responsabilidade de remover da pilha os valores que porventura ali armazene, antes de terminar.

3.4.2 Utilização da pilha na preservação de registradores

Quando se chama uma sub-rotina, é necessário saber quais registradores poderão ser modificados durante seu processamento. Se, imediatamente antes da chamada, algum desses registradores contiver um valor que se deseja conservar para depois do retorno, este precisa ser armazenado em algum outro lugar e depois recuperado. A pilha se presta de forma bastante prática para esta finalidade, e por isso é comum encontrar trechos de código como:

```
PUSH PSW      ; salva PSW na pilha
PUSH B         ; salva par BC
CALL 1234H     ; sub-rotina que altera BC e PSW
POP B          ; recupera par BC
POP PSW        ; recupera PSW
```

Note que a ordem em que se recuperam os registradores deve ser a inversa daquela em que são salvos, a menos que se deseje que ocorra troca de valores.

4 Programação do 8085

4.1 Assembler e linguagem Assembly

Um *assembler* é uma ferramenta de software – um programa – que simplifica a tarefa de criar programas. Recebe como entrada um arquivo-texto com as instruções do programa, denominado de *arquivo-fonte*, e gera um *arquivo-objeto*, que contém os códigos binários correspondentes.

A fim de que o assembler possa localizar e seguir as instruções do arquivo-fonte, o programador deve seguir certas regras. O conjunto dessas regras determina a sintaxe a ser empregada na escrita do arquivo e define, assim, uma linguagem de programação. Uma linguagem destinada a trabalhar com as instruções de um processador é denominada *linguagem Assembly*.

Em Assembly, as palavras-chave que representam as instruções são mnemônicos que correspondem diretamente aos códigos hexadecimais do processador. Uma vez que cada processador tem um conjunto de mnemônicos próprio, existe uma linguagem Assembly para cada processador.

Os mnemônicos são abreviaturas que lembram a função da instrução que representam, tais como MOV (do inglês *move*, para as instruções de movimentação de dados) ou JMP (do inglês *jump*, para os desvios).

A linguagem Assembly, no entanto, não pode se limitar aos mnemônicos, porque, além de dizer quais são as instruções que deverão compor o programa, é preciso dizer também onde elas devem ficar, estabelecer sinônimos para constantes e criar e utilizar variáveis. Por causa disso, existem, além dos mnemônicos, as *diretivas de montagem*, ou simplesmente *diretivas*, ou ainda *pseudo-instruções*. Estas são palavras-chave permitidas dentro da linguagem, mas que não geram código executável, porque não correspondem a qualquer instrução do processador – apenas ajudam a determinar a tarefa do assembler. As diretivas mais comuns são discutidas na seção 4.5.

4.2 Emprego da linguagem Assembly

Do ponto de vista do programador que já escreveu programas utilizando diretamente códigos hexadecimais ou binários, o uso da linguagem Assembly é muito vantajoso, por facilitar e agilizar a produção de programas.

Do ponto de vista de quem utiliza linguagens de alto nível, o emprego do Assembly tem vantagens e desvantagens. Por um lado, constitui uma forma de explorar os recursos do processador que não são acessíveis de outra forma e dá ao programador controle total sobre suas operações. Por outro lado, em geral é preciso escrever várias instruções Assembly para conseguir o mesmo resultado de uma instrução de linguagem de alto nível e também cuidar de tarefas que, de outro modo, seriam automatizadas pelo compilador. É mais fácil cometer erros e estes são, com frequência, difíceis de localizar. Isto aumenta os custos e o tempo de desenvolvimento, e por isso é bastante raro encontrar programas escritos totalmente em Assembly. Em geral, os programas são escritos quase que exclusivamente em linguagens de alto nível, e o Assembly é empregado apenas em partes específicas, por razões de otimização de espaço de memória ou de velocidade de execução, ou ainda para fazer acesso a hardware externo.

Apesar desse uso relativamente restrito, o estudo da programação em baixo nível é extremamente valioso para programar em outras linguagens, porque certos elementos destas só podem ser compreendidos com clareza quando se consegue interpretar seu significado em termos de nível mais baixo.

O material apresentado também permite reunir conhecimentos de áreas que normalmente são tratadas em separado, como sistemas digitais e técnicas de programação. Esta integração é fundamental para que se possa compreender o funcionamento de um sistema computacional em todos os níveis e fornece as bases para o raciocínio necessário em análise, projeto e teste de sistemas.

4.3 Instruções do 8085

Esta seção apresenta algumas características das instruções do 8085 que facilitam sua compreensão e memorização.

O conjunto completo de instruções é apresentado no anexo 1, e para uma descrição detalhada de cada instrução recomenda-se consultar [Inte77] ou [Visc82].

4.3.1 Formato

As instruções consistem de dois campos: o campo de código (*opcode*) e o campo dos operandos. O opcode diz o que a instrução faz, e por isso está sempre presente e é o primeiro byte de todas as instruções. Os operandos podem ocupar até dois bytes, de modo que o comprimento total de uma instrução do 8085 varia de um a três bytes. Por exemplo, a instrução RLC não necessita de maiores informações para saber que deve rotacionar o conteúdo do acumulador; é uma instrução de 1 byte. Já a instrução CPI precisa de mais um byte para comparar com o acumulador, e a instrução LXI B,<*dado16*> precisa de dois bytes para formar o dado de 16 bits.

Quase todas as instruções fazem referência a um ou mais registradores do processador, identificados pelas letras A, B, C, D, E, H e L. Algumas instruções referenciam esses registradores aos pares, agrupando-os da seguinte forma: B com C, D com E, H com L e A com os flags. Esses pares são identificados, respectivamente, por: B, D, H e PSW.

A letra M é utilizada para referenciar indiretamente o conteúdo de uma posição de memória. O endereço utilizado pelas instruções que usam esse mecanismo é o conteúdo do par HL no instante da sua execução. Este mecanismo é útil na varredura de tabelas ou vetores.

Os dados podem ser de 8 ou de 16 bits, dependendo do envolvimento de um único registrador ou de um par de registradores. Os endereços têm 16 bits quando se referem a uma posição de memória e 8 bits quando se referem a um endereço de E/S.

Mnemônicos com a letra X referem-se a operações realizadas sobre um par de registradores ou um registrador de 16 bits. A instrução DCX B, por exemplo, decrementa o par BC e a instrução INX SP incrementa o stack pointer.

Valores numéricos contidos nas instruções são denominados *dados imediatos*. Estes podem ser expressos em diferentes sistemas de numeração, indicados por uma letra, adicionada à direita do dado: H (hexadecimal), D ou T (decimal), Q (octal) ou B (binário). Números sem indicador são considerados decimais. Os dados expressos em hexadecimal devem iniciar sempre com um dígito numérico (por exemplo, FACA H deve ser escrito 0FACA H). Caso contrário, não são interpretados como símbolos numéricos, mas como labels ou sinônimos definido pelo usuário. Constantes em código ASCII podem ser representadas entre apóstrofes (por exemplo, 'von Neumann').

4.3.2 Classificação

As instruções do 8085 classificam-se em quatro tipos:

- *instruções lógicas e aritméticas*, que permitem efetuar operações como adição, subtração, deslocamento de bits, complemento, incremento e ajuste decimal (AND, OR, XOR, ADI, ADD, SUB, ANI, RAL, DAA, etc.);
- *instruções de transferência de dados*, que permitem transferir dados entre registradores, memória e periféricos (MOV, IN, LDA, PUSH, etc.);
- *instruções de desvio*, que permitem alterar a sequência normal de execução do programa, através de chamadas e retornos de sub-rotinas ou saltos (JMP, CALL, RET, JNZ, RNZ, CC, RST, etc.);
- *instruções de controle do processador*, que permitem controlar seu estado de execução, tais como a retenção (HLT) ou a habilitação de interrupções.

4.4 Formato de uma linha de código Assembly

Cada linha de um programa em Assembly pode apresentar os seguintes campos: label (rótulo), mnemônico, operandos e comentário. É importante salientar que os labels devem ser escritos sempre a partir da primeira coluna de cada linha, e que instruções e diretivas não podem começar na primeira coluna.

Os campos são delimitados através de alguns símbolos próprios: um ou mais espaços em branco permitem separar dois campos; dois pontos (:) permitem especificar o final do campo de um label; o ponto e vírgula (;) marca o início do campo de comentários e a vírgula (,) permite separar dois operandos. Os comentários encerram-se automaticamente no final da linha. Um exemplo de linha com todos esses elementos é:

```
START:  MVI A,00H           ; Inicializa contador
```

Neste exemplo, START é um label e MVI é o mnemônico da instrução. Os operandos são A e 00H.

O uso dos labels aumenta a clareza e a facilidade de manutenção dos programas e livra o programador do cálculo de endereços para colocar nas instruções. Por exemplo, para fazer um desvio para uma determinada linha, basta etiquetá-la com um label e depois usar este label como operando da instrução de desvio:

```
LOOP:   MOV A,M
        INX H
        CMP L,B
        JNZ LOOP           ; Instrução de desvio
```

O assembler se encarrega de calcular o endereço da linha de destino, etiquetada com LOOP. Por exemplo, se essa linha cair no endereço 2010H, o assembler gerará na última linha do trecho a instrução JMP 2010H e corrigirá este endereço sempre que o trecho mudar de lugar. Se o programador optasse por não utilizar o label, teria que fazer essa correção manualmente.

4.5 Diretivas do assembler

As *diretivas de montagem* permitem definir símbolos, estabelecer endereços específicos para determinados trechos do programa (como por exemplo o endereço inicial) ou ainda reservar áreas de memória para variáveis globais. É importante lembrar sempre que as diretivas não fazem parte do conjunto de instruções do processador e que não geram código binário.

As principais diretivas utilizadas em programas para o 8085 são descritas a seguir, acompanhadas de exemplos.

<nome> DB <valor inicial>

A diretiva DB (*define byte*) permite reservar áreas de memória e identificar essas áreas com um nome. O nome utilizado na diretiva torna-se um sinônimo para o endereço do primeiro byte reservado. Uma vez que é possível referenciar esse nome a partir de qualquer ponto do programa, o espaço reservado é conhecido como uma *variável global*. As diretivas citadas permitem ainda atribuir valores iniciais a essas variáveis, conforme ilustram os exemplos a seguir.

```
V1      DB 200
```

cria uma variável global de 1 byte, cujo valor inicial é 200. O símbolo V1 passa a ser um sinônimo para o endereço da variável. A escolha deste endereço pode ser feita pelo programador através da diretiva ORG (v. adiante), ou então deixada a cargo do assembler. Neste caso, a variável será simplesmente colocada na sequência de tudo que o assembler vinha criando até ali.

A linha a seguir cria uma variável global de 3 bytes, inicializados conforme a sequência dada; pode ser interpretada como uma tabela, cujo primeiro elemento reside no endereço V2:

```
V2      DB 1, 2, 3
```

A linha seguinte define uma cadeia de caracteres, ou *string*, ocupando 3 bytes:

```
P1      DB 'MIC'
```

Os valores iniciais dos bytes criados pela linha acima correspondem ao código ASCII de cada caracter, respectivamente 4DH, 49H e 43H.

ORG <endereço>

O símbolo ORG vem da palavra inglesa *origin*. Esta diretiva estabelece o endereço em que o assembler colocará a próxima instrução ou variável global. É utilizada para determinar o endereço inicial do programa e a posição de outros trechos de código, quando se deseja que estes residam num endereço bem determinado, como no caso dos tratadores de interrupção. Quando seguida por uma das diretivas de reserva de área de memória vistas acima, serve para determinar a posição de uma área de dados. Um programa pode conter várias diretivas ORG.

O endereço pode ser fornecido na forma absoluta (um valor, geralmente em hexadecimal) ou relativa à posição da linha atual, cujo endereço é representado pelo símbolo \$. Por exemplo,

```
ORG 2000H
```

refere-se à posição de memória cujo endereço é 2000H e

```
ORG $+100
```

transfere o ponto de continuação da montagem 100 bytes adiante.

<sinônimo> EQU <valor>

Permite associar um nome a um valor. Por exemplo,

```
LIMIT EQU 0FFH
```

permite utilizar o símbolo LIMIT como sinônimo para o valor 0FFH em qualquer parte do programa.

O uso de sinônimos é altamente recomendável por duas razões principais:

- aumentam a qualidade do programa por facilitarem seu entendimento;
- facilitam a manutenção se o valor precisar ser modificado. Neste caso, basta alterar a diretiva para redefinir o valor; se não se utilizasse o sinônimo, o valor utilizado teria que ser modificado em todos os pontos do programa onde tivesse sido colocado.

END

Indica o fim do programa fonte ao assembler, que ignora qualquer linha escrita após esta diretiva. Sua presença é obrigatória no final do programa, sob pena de o assembler se perder. É importante lembrar de colocar espaços no início da linha, pois trata-se de uma diretiva, e não de um label.

A tabela 4.1 resume as diretivas apresentadas.

Diretiva	Função	Exemplo
ORG	Definir endereço	ORG 2000H
EQU	Criar sinônimo	TAMANHO EQU 25H
DB <valor>	Criar variável	COUNT DB 00H
DB <valor>,...	Criar vetores	TABELA DB 15H, 22H, ...
DB <string>	Criar strings ASCII	MSG DB 'Mensagem 1'
END	Final de programa	END

Tab. 4.1 – Diretivas da linguagem Assembly do 8085

4.6 Etapas do desenvolvimento em Assembly

Um programa em desenvolvimento sofre diversas transformações, desde o momento em que é editado até chegar à sua forma executável. Normalmente, estas transformações são realizadas com o auxílio de ferramentas dedicadas. As etapas que compõem o processo de desenvolvimento são, tipicamente:

- *geração do código-fonte*, em que se escreve, utilizando um editor de texto, o programa em linguagem Assembly; este pode compreender um ou mais *arquivos-fonte*;
- *tradução do código-fonte*, em que o assembler gera um arquivo-objeto para cada arquivo-fonte;
- *edição de ligações ou “linkagem”*, em que uma ferramenta chamada *linker* une os diversos arquivos-objeto num arquivo utilizável para execução ou simulação;
- *criação ou manutenção de bibliotecas de código-objeto*, onde se armazenam sub-rotinas passíveis de reaproveitamento em outros programas.

4.7 Exemplo de programa em Assembly

O programa a seguir é previsto para execução no Abacus, descrito no capítulo 10. LETECLA e MOSTRAD são sub-rotinas simuladas em ROM que permitem, respectivamente, ler um dígito hexadecimal do teclado e mostrar o conteúdo do par de registradores DE no campo de endereços do display. Seus endereços default aparecem na figura 10.9.

O programa deve efetuar a multiplicação de dois números hexadecimais de 8 bits e apresentar o resultado, que terá, no máximo, quatro dígitos.

O programa principal

O programa principal deve ser escrito de forma que sua leitura permita compreender imediatamente o que o programa todo faz. Por isso, serviços auxiliares, como a leitura dos dados ou sua apresentação no display, devem ser delegados para sub-rotinas, para não desviar a atenção de quem tenta compreender o programa. Desta forma, o desenvolvimento do programa principal serve também de guia para determinar quais são as demais sub-rotinas que constituirão o programa.

Os comentários são escritos sem caracteres acentuados, porque estes criam problemas para o assembler:

```
LETECLA EQU 02E7H      ;
MOSTRAD EQU 0363H      ; sinonimos para sub-rotinas em ROM
                ORG 2000H      ; endereco inicial do programa
INICIO: LXI SP,20C0H    ; define inicio da pilha
LOOP0:  CALL LE_DADO      ; leitura de N1 (acumulador)
        MOV B,A           ; copia N1 para o registrador B
        CALL LE_DADO      ; leitura de N2 (acumulador)
        MOV C,A           ; copia N2 para o registrador C
        CALL MULT         ; DE = C * B
        CALL MOSTRAD      ; apresenta resultado
        JMP LOOP0         ; reinicia
```

A leitura de dados

O programa principal acima precisa de uma sub-rotina chamada LE_DADO, para fazer a leitura de cada um dos números hexadecimais de dois dígitos. Esta deve levar em consideração os seguintes pontos:

- a sub-rotina LETECLA, que pode ser utilizada para ler o teclado, lê um dígito de cada vez; por isso, precisa ser chamada duas vezes para ler um número de dois dígitos;
- o primeiro dos dois dígitos hexadecimais a ser lido é o dígito mais significativo. Este deve ser multiplicado por 16 e transferido do acumulador para um outro registrador, para não ser apagado na chamada seguinte de LETECLA;
- o número hexadecimal de dois dígitos será obtido através de uma operação OU entre o registrador que armazena o dígito mais significativo, já multiplicado por 16, e o dígito menos significativo;
- o número obtido deve ser retornado no acumulador.

Com isso, a sub-rotina de leitura fica assim:

```

LE_DADO:CALL LETECLA      ; le o dígito mais significativo
          RLC
          RLC
          RLC
          RLC              ; multiplica-o por 16
          MOV E,A          ; dígito mais significativo em E
          CALL LETECLA     ; le dígito menos significativo
          ORA E            ; compoe numero de dois dígitos
          RET              ; retorna com numero em A

```

A sub-rotina de multiplicação

A sub-rotina MULT deve multiplicar dois números hexadecimais de 8 bits, contidos em B e C, e colocar o resultado desta multiplicação no par DE.

Pode-se começar criando uma sub-rotina que multiplica dois números de dois dígitos, n_1 e n_2 , com resultado de dois dígitos (válida somente para os casos em que o resultado da multiplicação é inferior a 255).

A multiplicação de n_1 e n_2 será efetuada somando-se n_2 vezes o operando n_1 . Considerando que os registradores B e C contêm, respectivamente, os operandos n_1 e n_2 , o trecho de código a seguir faz a multiplicação e apresenta o resultado no registrador E:

```

MULT:    MVI A,00          ; inicializacao
LOOP1:   ADD B              ; acrescenta valor de N1 ao acumulador
          DCR C             ; decrementa o conteudo de C
          JNZ LOOP1         ; repete a soma N2 vezes
          MOV E,A           ; transfere o resultado para E
          RET               ; retorno com resultado em E

```

A seguir, esta sub-rotina deve ser estendida para tratar resultados com 4 dígitos hexadecimais, incluindo, assim, a possibilidade de o resultado ser superior a 255. Para tanto, basta incrementar o registrador D (que vai apresentar os dois dígitos mais significativos) cada vez que o resultado da adição no acumulador ultrapassar 255, conforme indicado pelo Carry Flag:

```

MULT:    MVI D,00          ; inicializacoes
          MVI A,00
LOOP1:   ADD B              ; soma N1 ao acumulador
          JNC CONT          ; "vai 1"?
          INR D              ; incrementa D se a soma passa de FF
CONT:    DCR C
          JNZ LOOP1         ; repete a soma N2 vezes
          MOV E,A           ; transfere o resultado para E
          RET               ; retorno com resultado no par DE

```

O programa completo consiste, então, do programa principal e das sub-rotinas MULT e LE_DADO. O arquivo-fonte precisa conter, ainda, uma diretiva END, no final. Consulte o capítulo 10 para continuar com a montagem e a execução no Abacus.

4.8 Outros trechos de código

Esta seção apresenta alguns trechos de código típicos de programas Assembly. Seu estudo é de grande valia para aumentar as habilidades de programação, visto que as tarefas discutidas aparecem com grande frequência na programação de qualquer microprocessador. Os trechos apresentados dão uma idéia de como compiladores de linguagens de alto nível implementam os comandos de tomada de decisão (if) e de repetição (while, for).

4.8.1 Tomando decisões

Tomar uma decisão em Assembly é sempre um processo de duas etapas: uma comparação seguida de um desvio condicional. A operação de comparação tem por função preparar os flags para a operação de desvio. As diferentes instruções de desvio condicional podem ser utilizadas para implementar diversos critérios de decisão.

Os exemplos a seguir exploram essas possibilidades. Em todos os exemplos que comparam dois números, assume-se que estes se chamam X e Y e que se encontram, respectivamente, nos registradores A e B.

Execução de um bloco de instruções se $X = Y$

```
CMP B           ; seta flags de acordo com X - Y
JNZ CONT        ; salta se  $X \neq Y$ 
...             ; instruções executadas se  $X = Y$ 
CONT:           ; ponto de continuação do programa
```

Execução de um bloco de instruções se $X \neq Y$

```
CMP B           ; seta flags de acordo com X - Y
JZ CONT         ; salta se  $X = Y$ 
...             ; instruções executadas se  $X \neq Y$ 
CONT:           ; ponto de continuação do programa
```

Execução de um bloco de instruções se $X \geq Y$

```
CMP B           ; seta flags de acordo com X - Y
JC CONT         ; salta se  $X < Y$ 
...             ; instruções executadas se  $X \geq Y$ 
CONT:           ; ponto de continuação do programa
```

Execução de um bloco de instruções se $X > Y$

```
CMP B          ; seta flags de acordo com X - Y
JC CONT        ; salta se X < Y
JZ CONT        ; salta se X = Y
...            ; instruções executadas se X > Y
CONT:          ; ponto de continuação do programa
```

Execução de um bloco de instruções se $X \leq Y$

```
CMP B          ; seta flags de acordo com X - Y
JZ OK          ; salta se X = Y
JNC CONT       ; salta se X ≥ Y
OK:            ; instruções executadas se X ≤ Y
CONT:          ; ponto de continuação do programa
```

Execução de um bloco de instruções se $X < Y$

```
CMP B          ; seta flags de acordo com X - Y
JNC CONT       ; salta se X ≥ Y
...            ; instruções executadas se X < Y
CONT:          ; ponto de continuação do programa
```

O próximo exemplo implementa uma construção do tipo `if... then... else`, comum em linguagens de alto nível.

Execução de um bloco se $X = Y$ e de outro se $X \neq Y$

```
CMP B          ; seta flags de acordo com X - Y
JNZ DIFF       ; salta se X ≠ Y
...            ; instruções executadas se X = Y
JMP CONT       ; salto para isolar o bloco seguinte
DIFF:          ; instruções executadas se X ≠ Y
CONT:          ; ponto de continuação do programa
```

Note o uso das instruções JC e JNC nos exemplos em que é preciso determinar se X é maior ou menor do que Y. O carry flag indica se houve “empréstimo 1” na subtração feita pela instrução de comparação. Se houve (CY = 1), então o valor do acumulador é menor do que o do registrador usado; se não houve (CY = 0), então é maior ou igual.

Atenção para o código a seguir, que emprega a instrução JM no lugar de JC para executar um bloco se $X \geq Y$:

```

CMP B          ; seta flags de acordo com X - Y
JM CONT        ; desejamos saltar se X < Y

...           ; instruções a executar se X ≥ Y
CONT:          ; ponto de continuação do programa

```

Este código também funciona, mas apenas enquanto $A - B < 80H$. Se esta condição não estiver satisfeita, a instrução de comparação seta o sign flag e o código passa a se comportar de forma contrária à esperada. Por exemplo, para $A = 88H$ e $B = 02H$, o resultado da subtração interna é $86H$, que é interpretado como um número negativo. Assim, o código acima consideraria $88H$ menor do que $02H$. Este problema não acontece ao se usar o carry flag, e é por isso que o correto é utilizar as instruções JC e JNC para fazer as comparações. As instruções JM e JP devem ser utilizadas quando for necessário determinar se o resultado de uma operação é um número positivo ou negativo.

Os próximos exemplos implementam condições duplas de teste. Para tanto, considere que se trabalha agora com quatro números, X, Y, Z e W, contidos respectivamente nos registradores A, B, C e D.

Execução de um bloco de instruções se $X = Y$ E $W = Z$

```

CMP B          ; seta flags de acordo com X - Y
JNZ CONT       ; se falhar a primeira condição, não
               ; é preciso testar a segunda
MOV A,C        ; assume-se que X pode ser destruído
CMP D          ; seta flags de acordo com W e Z
JNZ CONT       ; salta se falhar a segunda condição

...           ; instruções executadas se X = Y
CONT:          ; ponto de continuação do programa

```

Execução de um bloco de instruções se $X = Y$ OU $W = Z$

```

CMP B          ; seta flags de acordo com X - Y
JZ OK          ; se OK na primeira, não é preciso
               ; testar a segunda
MOV A,C        ; assume-se que X pode ser destruído
CMP D          ; seta flags de acordo com W e Z
JNZ CONT       ; salta se falhar a segunda condição

OK:           ; instruções executadas se X = Y
CONT:          ; ponto de continuação do programa

```

4.8.2 Repetições

As repetições são muito comuns em todos os tipos de programas. Nas linguagens de alto nível, as estruturas de repetição implementadas pelos exemplos a seguir correspondem a comandos como `for`.

Loop infinito

```
LOOP:  ...  ; bloco de instruções a repetir indefinidamente
        JMP LOOP
```

Loop repetido C vezes; se $C = 0$, é executado 256 vezes

```
LOOP:  ...                ; bloco de instruções a repetir
                        ; não deve alterar o valor de C
        DCR C              ; decrementa C e seta flags;
        JNZ LOOP
```

Loop repetido C vezes; se $C = 0$, não é executado

```
        MVI A,00H
        CMP C
        JZ CONT
LOOP:   ...                ; bloco de instruções a repetir
                        ; não deve alterar o valor de C
        DCR C              ; decrementa C e seta flags;
        JNZ LOOP
CONT:   ...                ; continuação do programa
```

As condições apresentadas no item 4.8.1 podem também aparecer como critérios de encerramento das repetições. Assim, é possível repetir um bloco de instruções enquanto uma certa condição for verdadeira, como por exemplo $X > Y$ ou $X = Y$. É isso que cria um comando `while`.

4.8.3 Casos particulares de multiplicação

Quando o resultado da multiplicação de dois números não ultrapassa o tamanho do acumulador (FFH, no caso dos processadores de 8 bits), é possível efetuar a multiplicação por deslocamento de bits ao invés de empregar a repetição da soma. O método baseia-se no fato de que, dentro dessas condições, a operação RLC acrescenta um zero à direita do multiplicando e, portanto, multiplica-o por 2. Aplicando-se esta operação várias vezes e guardando os resultados intermediários de forma conveniente, pode-se efetuar

a multiplicação de dois números quaisquer, conforme ilustram os exemplos a seguir. Em todos eles, o valor inicial do acumulador é representado pela letra x.

Multiplicação por 16

Este é um dos casos mais simples, pois $16 = 2^4$. Assim, basta a sequência

```
RLC          ; A = 2x
RLC          ; A = 4x
RLC          ; A = 8x
RLC          ; A = 16x
```

para multiplicar por 16 o conteúdo do acumulador. Em geral, a multiplicação por 2^n se faz com n operações RLC ou $8-n$ operações RRC.

Multiplicação por 10

A multiplicação por 10 se faz com auxílio de um registrador intermediário:

```
RLC          ; A = 2x
MOV B,A      ; B = 2x
RLC          ; A = 4x
RLC          ; A = 8x
ADD B        ; A = 8x + 2x = 10x
```

Multiplicação por 7

Primeira solução:

```
MOV B,A      ; B = x
RLC          ; A = 2x
ADD B        ; A = 3x
RLC          ; A = 6x
ADD B        ; A = 7x
```

Segunda solução (somente viável se $8x < 100H$):

```
MOV B,A      ; B = x
RLC          ;
RLC          ;
RLC          ; A = 8x
SUB B        ; A = 7x
```


5 Interrupções do 8085

As interrupções, assim como as sub-rotinas, constituem um mecanismo de desvio do fluxo de processamento de um programa. A razão da sua existência é um tipo de problema que acontece em inúmeras situações práticas: os casos em que um programa deve executar uma certa tarefa em resposta à ocorrência de um evento externo. Exemplos de eventos externos são o pressionar de um botão, a recepção de um dado numa porta serial, a atuação de um sensor ou a ocorrência de um alarme.

5.1 Interrupções x sub-rotinas

A princípio, parece razoável escrever o código que deve ser executado por ocasião da ocorrência do evento em questão na forma de uma sub-rotina, que seria chamada quando o evento acontecesse. Em princípio, a idéia é essa mesmo, mas falta um mecanismo que notifique o programa da ocorrência do evento. Sem isso, o programa não pode saber quando os eventos acontecem e portanto não sabe quando deve executar a sub-rotina.

Existem duas abordagens básicas para resolver este problema: a primeira, denominada varredura (ou, em inglês, *polling* – pronuncia-se “póling”), consiste em verificar periodicamente uma porta de entrada, esperando que o evento externo seja sinalizado para então executar a sub-rotina correspondente. Este princípio é simples, mas o fato de a CPU ter que ficar fazendo a varredura periodicamente é, em geral, indesejável.

A segunda abordagem, que é a que nos interessa aqui, é o uso de uma interrupção. Para permitir a implementação deste mecanismo, o 8085 conta com cinco entradas de sinal que podem ser ativadas por componentes periféricos para a notificação de eventos externos. Em resposta a um estímulo numa dessas entradas, a CPU desvia o processamento do programa para um trecho de código semelhante a uma sub-rotina, denominado *tratador de interrupção*, que executa as tarefas especificadas para a ocorrência do evento.

Neste ponto é interessante analisar as diferenças entre sub-rotinas e interrupções. Para tanto, convém lembrar que as sub-rotinas têm as seguintes características:

- são trechos de código localizados em endereços definidos pelo programador;
- o desvio para o código de uma sub-rotina acontece quando o processador executa uma instrução de desvio da forma *CALL <endereço de destino>*, em que é fornecido o endereço da sub-rotina chamada;
- o mecanismo de retorno consiste em salvar o endereço da instrução seguinte ao *CALL* na pilha e recuperá-lo quando da execução da instrução *RET*, que determina o retorno da sub-rotina.

As interrupções, por sua vez, não podem utilizar o mesmo tipo de mecanismo de chamada. Por definição, elas podem ocorrer a qualquer momento, não importa qual seja a instrução em execução. Isto significa que deve ser possível fazer o desvio sem a presença de uma instrução de desvio, o que implica ainda que não se pode fornecer um endereço de destino da forma como se faz nas chamadas de sub-rotinas.

Existem diversas soluções para este problema. A mais simples consiste em atribuir a cada sinal de interrupção um endereço fixo no projeto do processador. Das cinco entradas de interrupção do 8085, quatro têm endereços fixos, conforme a tabela 5.1.

Sinal	Endereço
TRAP	0024H
RST 5.5	002CH
RST 6.5	0034H
RST 7.5	003CH

Tab. 5.1 – Endereços de desvio das interrupções do 8085

Além destas, existem os sinais *INTR* (*interrupt request*) e *INTA* (*interrupt acknowledge*), concebidos para permitir a comunicação do 8085 com o controlador de interrupções 8259, da Intel. Este componente tem oito entradas de interrupção, às quais podem ser conectados quaisquer circuitos periféricos que precisem interromper o processador. Ao receber uma solicitação de interrupção em uma de suas entradas, o 8259 ativa o sinal *INTR* do 8085. Este termina de executar a instrução em andamento e então responde com o sinal *INTA*, sinalizando ao 8259 que o pedido de interrupção pode ser atendido. Em seguida, o controlador de interrupção coloca no barramento de dados o código de uma instrução, que o 8085 executará em resposta à interrupção. Esta

instrução pode ser escolhida pelo programador e pode ser qualquer instrução de comprimento igual a 1 byte do conjunto de instruções do 8085. A escolha é feita mediante programação do 8259 pelo próprio 8085, durante a inicialização do programa. Desta forma, cada entrada de interrupção do 8259 pode ter um efeito diferente sobre o programa. Geralmente, as instruções escolhidas são do tipo RST (v. seção 5.4), resultando na chamada de um tratador para a interrupção em questão.

O tratamento de interrupções no 8085 tem as seguintes características:

- os tratadores associados aos sinais RST 5.5, RST 6.5, RST 7.5 e TRAP são trechos de código localizados em endereços fixos, definidos no projeto do processador; apenas para aqueles associados ao sinal INTR existe uma certa liberdade de escolha, conforme os endereços de desvio das instruções RST 0 a RST 7;
- o desvio para o tratador acontece logo após o término da execução da instrução em andamento (a instrução que está sendo processada no instante em que a interrupção é detectada), desde que a interrupção solicitada esteja habilitada, de acordo com o mecanismo da seção 5.2;
- o mecanismo de retorno é idêntico ao das sub-rotinas; o endereço da instrução seguinte à última instrução executada antes do desvio é armazenado na pilha e o código do tratador terminando por uma instrução RET.

5.2 Habilitando e inibindo interrupções

Há situações em que as interrupções não são bem-vindas. Por exemplo, em um programa que controla um processo em tempo real, pode haver momentos em que toda a capacidade de processamento deve estar voltada à obtenção de um resultado dentro do menor tempo possível, e neste caso qualquer interrupção seria prejudicial. Outro caso, clássico, é a interrupção do próprio tratador de interrupção, que pode acontecer se uma segunda interrupção for sinalizada durante o tratamento de outra, ocorrida um pouco antes. Este tratamento de interrupções de forma recursiva é possível, mas em geral muito complicado e quase sempre dispensável.

Os microprocessadores têm, por isso, mecanismos que permitem inibir as interrupções. No 8085, este controle é feito através da *máscara de interrupções* (*interrupt mask*), um registrador que permite habilitar ou inibir individualmente as interrupções RST 5.5, 6.5 e 7.5. Este registrador contém também os bits de controle dos sinais de comunicação serial do processador (bits 6 e 7), embora

estes nada tenham a ver com as interrupções. A habilitação das interrupções é feita através da instrução SIM (*Set Interrupt Mask*), que transfere para a máscara um valor previamente colocado no acumulador. A figura 5.1 mostra como o valor colocado no acumulador é interpretado pela instrução SIM.

7	6	5	4	3	2	1	0
E/S serial	Habilita serial		Reset FF 7.5	Habilita máscara	RST 7.5	RST 6.5	RST 5.5

Fig. 5.1 – A máscara de interrupções vista pela instrução SIM

Os bits 0, 1 e 2 inibem (bit igual a 1) ou habilitam (bit igual a zero) as interrupções 5.5, 6.5 e 7.5, respectivamente, mas só têm efeito se o bit 3 estiver setado. Este bit é colocado em zero quando se deseja modificar o valor da máscara de interrupções apenas para controlar a porta de saída serial, sem interferir no estado atual das interrupções.

Para compreender o papel do bit 4, é preciso saber que cada uma das entradas RST 5.5, 6.5 e 7.5 é dotada de um flip-flop, que memoriza um pedido de interrupção feito enquanto a interrupção correspondente estiver desabilitada. Desta forma, o pedido fica pendente até que a interrupção volte a ser habilitada. Às vezes, um pedido pendente pode se tornar obsoleto e por isso pode ser preferível descartá-lo. Colocar o bit 4 em 1 reseta o flip-flop da entrada RST 7.5, descartando a solicitação pendente. As solicitações das outras entradas não podem ser canceladas.

O bit 5 não é utilizado. O bit 6 habilita ou inibe a escrita na porta serial, que só reage se este bit estiver em 1. Este bit é colocado em zero quando se deseja modificar o conteúdo da máscara de interrupções apenas para controlar as interrupções, sem interferir na porta serial. É o dual do bit 3.

O bit 7 é o dado (0 ou 1) enviado para fora do processador através do pino SOD quando a máscara de interrupções for escrita com o bit 6 em 1.

A seqüência

```
MVI A,18H
SIM
```

por exemplo, cancela um pedido pendente da interrupção 7.5, se houver, e habilita as interrupções 5.5, 6.5 e 7.5, sem interferir na porta serial.

Além dos bits 0 a 2 da máscara de interrupções, o 8085 apresenta uma espécie de chave geral, que permite habilitar ou inibir todas as interrupções de uma só vez, com exceção da interrupção TRAP. Este controle é feito pelas instruções EI (*enable interrupts*) e DI (*disable interrupts*), que controlam o flag IE (*Interrupt Enable*). Os bits correspondentes às interrupções 5.5, 6.5 e 7.5 na máscara de interrupções não são afetados pelas instruções EI e DI, mas as interrupções

habilitadas pela máscara só estarão de fato habilitadas se o flag IE estiver em 1. A principal vantagem das instruções EI e DI está em permitir que se ative e desative o mecanismo de interrupção sem que seja necessário salvar e restaurar a configuração da máscara.

O flag IE é automaticamente resetado nas seguintes condições:

- após um RESET do processador;
- após atendimento de uma interrupção (isto evita que múltiplos pedidos sejam atendidos recursivamente).

Desta forma, todo tratador de interrupção inicia com as interrupções inibidas, o que corresponde à necessidade da maioria dos tratadores. Se for necessário habilitar as interrupções durante o tratamento, o tratador deverá incluir uma instrução EI. Normalmente, esta é a última instrução executada dentro do tratador antes do RET, para que as interrupções voltem a ser habilitadas após o término do processamento do tratador.

A máscara de interrupções também pode ser lida. Para tanto, utiliza-se a instrução RIM (*Read Interrupt Mask*), que transfere o estado atual da máscara para o acumulador. A interpretação dos bits após a leitura da máscara difere da descrição anterior e é mostrada na figura 5.2.

7	6	5	4	3	2	1	0
E/S serial	RST 7.5 ?	RST 6.5 ?	RST 5.5 ?	IE	RST 7.5	RST 6.5	RST 5.5

Fig. 5.2 – A máscara de interrupções vista pela instrução RIM

Os bits 0 a 2 têm a mesma interpretação da instrução SIM; o bit 3 é o flag IE (*Interrupt Enable*), controlado pelas instruções EI e DI. Os bits 4 a 6 permitem saber o estado dos flip-flops das interrupções 5.5 a 7.5, de modo que um programa em execução tem como descobrir se há pedidos pendentes enquanto as interrupções estão desabilitadas.

A máscara de interrupções não faz referência às interrupções INTR e TRAP. No primeiro caso, o 8085 pode controlar a habilitação de cada uma das oito interrupções do controlador externo, programando-o convenientemente. A interrupção TRAP, por sua vez, não é mascarável, e portanto as solicitações feitas nesta entrada são sempre atendidas imediatamente. Por isso, ela é normalmente utilizada em situações críticas. Por exemplo, pode-se projetar uma fonte de alimentação que avise o sistema através dessa interrupção que a rede de energia elétrica caiu, e que a alimentação só pode ser mantida por mais alguns instantes, enquanto se descarregam os capacitores da fonte. Qualquer que seja a tarefa em execução, em geral não haverá justificativa para

continua-la. É mais sensato desviar o processamento para o tratador da interrupção TRAP, que pode, ao menos, salvar informações vitais ou tomar outras providências que evitem danos maiores ao sistema.

5.3 Prioridades das interrupções

O 8085 verifica a existência de pedidos de interrupção uma vez a cada ciclo de execução. Desta forma, pode acontecer que em um dado ciclo não haja qualquer interrupção pendente, mas que haja mais de um pedido aguardando atendimento no ciclo seguinte. Neste caso, o processador atende os pedidos de acordo com a seguinte prioridade: TRAP, RST 7.5, RST 6.5, RST 5.5 e INTR. A mesma prioridade é utilizada caso haja mais de um pedido pendente quando a execução de uma instrução EI volta a habilitar as interrupções.

É importante notar que as prioridades são utilizadas apenas nos casos em que há mais de um pedido pendente durante um mesmo ciclo de execução, e não para determinar se um tratador de interrupção tem prioridade sobre outro.

5.4 Interrupções de hardware e de software

As interrupções estudadas até aqui são conhecidas como *interrupções de hardware*, por estarem associadas a eventos do hardware externo. Existem ainda outras oito interrupções no 8085, denominadas *interrupções de software*, que são ativadas quando o processador executa uma das instruções RST n , onde n é um número de 0 a 7. Estas instruções se comportam como um CALL, mas com o endereço de destino pré-fixado em $8n$. A tabela 5.2 apresenta os endereços destas interrupções e inclui mais uma vez as interrupções RST 5.5, 6.5 e 7.5. Note que os endereços destas interrupções podem ser calculados multiplicando-se esses números por 8, fato que explica sua nomenclatura.

Interrupção	Endereço	Interrupção	Endereço
RST 0	0000H	RST 5	0028H
RST 1	0008H	RST 5.5	002CH
RST 2	0010H	RST 6	0030H
RST 3	0018H	RST 6.5	0034H
RST 4	0020H	RST 7	0038H
TRAP	0024H	RST 7.5	003CH

Tab. 5.2 – Interrupções de hardware e de software

5.5 Escrevendo tratadores de interrupção

O mecanismo de tratamento de interrupções descrito neste capítulo exige que se observem alguns pontos importantes quando se escreve um tratador de interrupção. Embora sejam apenas conseqüências do princípio de funcionamento desse mecanismo, os pontos discutidos a seguir não são, em geral, considerados óbvios e são freqüentemente motivo de erros graves de programação:

- *localização do tratador*: o programador deve se certificar de que o assembler colocará o tratador no endereço correspondente ao desvio previsto. Isto se faz, em geral, colocando uma diretiva `ORG <endereço>` antes do início do código do tratador. A ausência desta diretiva fará com que o assembler e o linkador tratem o código do tratador como o de outra sub-rotina qualquer, colocando-o numa posição diferente do desvio da interrupção;
- *preservação de registradores*: um tratador de interrupções deve salvar na pilha *todos* os registradores do processador que utiliza e restaurá-los antes de retornar. Negligenciar este passo e modificar registradores do processador dentro do tratador é extremamente perigoso. Como a interrupção pode acontecer, teoricamente, em qualquer ponto do programa, a modificação do conteúdo de um registrador pode ter conseqüências catastróficas. O mais grave é que esta modificação também pode ser inofensiva, quando a interrupção acontece durante a execução de um trecho de código que não utiliza o registrador que está sendo alterado. Se esta for a situação que acontece durante o teste do programa, este pode vir a ser aprovado com uma falha que pode se manifestar mais tarde, quando o sistema estiver em uso. As conseqüências são imprevisíveis, e a causa do problema é, em geral, muito difícil de localizar. Muitos programadores salvam sempre todos os registradores, mesmo que não os utilizem. Isto evita que uma modificação posterior do tratador, que venha a utilizar um registrador novo, introduza problemas;
- *a comunicação com o programa principal não pode ser feita através dos registradores*: uma vez que os registradores precisam ser preservados, pelas razões descritas acima, a comunicação entre o tratador e o programa principal, quando necessária, tem de ser feita de outra forma. O meio mais comum é definir uma variável global, que é alterada pelo tratador. O programa principal pode testar esta variável para descobrir o que o tratador fez;
- *o tratador deve ser curto*: em geral, as interrupções ficam inibidas durante o processamento do tratador. Isto tem a vantagem de simplificar o seu código (um tratador recursivo é sempre mais complexo, porque tem que prever que

pode ser interrompido por si mesmo a qualquer momento) e de limitar a profundidade da pilha (pois a cada chamada um novo endereço de retorno é empilhado, novos registradores são salvos, etc.). Por outro lado, a inibição das interrupções por períodos longos pode prejudicar a resposta do sistema a eventos externos, e por isso é recomendável que o código dos tratadores se limite a fazer o essencial, deixando para o corpo principal do programa as tarefas que puderem ser feitas fora do tratador;

☛ *as interrupções precisam estar habilitadas e a pilha não deve crescer indefinidamente:* sempre que um programa pretender utilizar interrupções, o código executado na inicialização precisa habilitá-las (no caso do 8085, através da configuração da máscara de interrupções e da instrução EI). Do contrário, nenhuma interrupção mascarável será atendida. É igualmente importante que o programador garanta um meio de reabilitar as interrupções depois que o tratador correspondente a cada uma tiver sido chamado, pois o atendimento de uma interrupção inibe as interrupções. Em geral, isto se faz colocando uma instrução EI no final de cada tratador, imediatamente antes do RET. Note que não é recomendável colocar qualquer instrução entre o EI e o RET, porque se uma nova interrupção acontecer depois do EI e antes do RET, a pilha receberá novamente o endereço de retorno e outros valores que o tratador deseje salvar e, se isto se repetir muitas vezes, o limite da pilha previsto pelo programador pode ser ultrapassado. Com o EI imediatamente antes do RET, isto jamais acontece, porque o EI habilita as interrupções apenas a partir da instrução que o segue (no caso, o RET), e esta, como vimos, é executada até o final antes do desvio para uma nova interrupção. Com isso, o endereço de retorno é retirado da pilha e esta não pode crescer indefinidamente;

☛ *o tratador não deve sujar a pilha:* assim como no caso das sub-rotinas, o retorno ao ponto de execução correto só é possível se, no instante da execução do RET, o SP estiver apontando para uma posição da pilha que contenha o endereço de retorno. Portanto, é responsabilidade do programador cuidar para que, no instante da execução do RET, a pilha esteja exatamente como estava quando o tratador foi chamado.

O capítulo 15 traz mais informações interessantes sobre o desenvolvimento de tratadores de interrupção.

6 O microprocessador 8086

6.1 Introdução

Conforme mencionado no capítulo 1, o 8086 faz parte de uma linhagem de microprocessadores, iniciada pela Intel com a primeira CPU integrada num único chip, o 4004. O 8086 sucedeu o 8085, e as principais modificações introduzidas neste novo produto foram:

- ampliação da capacidade de endereçamento de memória para 1 MB;
- arquitetura de 16 bits;
- novas instruções para a manipulação de seqüências de caracteres (strings);
- aritmética decimal completa.

Destes itens, são sem dúvida os dois primeiros que causam maior impacto na utilização do processador, e por isso as conseqüências dessas modificações são o principal tema do presente capítulo. As instruções de manipulação de strings são tratadas no capítulo 9, e o anexo 8 traz alguns detalhes sobre a divisão de números inteiros. O material apresentado serve de fundamento para explorar os títulos relacionados na bibliografia, que tratam o assunto de forma mais extensa.

6.2 8086 x 8088

O processador 8088 tem o mesmo conjunto de instruções e a mesma arquitetura interna do 8086 e é, portanto, um processador de 16 bits. A principal diferença entre os dois processadores está na largura do barramento de dados, que é de apenas 8 bits no 8088, no lugar dos 16 bits do 8086.

O lançamento do 8088 com esta característica foi um passo estratégico, visando tornar mais convidativa a migração de sistemas mais antigos, que utilizavam processadores de 8 bits, para a nova arquitetura de 16 bits. O barramento de dados do 8088 permitia que se aproveitassem, ao menos em

parte, projetos de sistemas que utilizavam processadores de 8 bits, como o 8085. Este era um fator importante numa época em que os custos de projeto de hardware, envolvendo recursos computacionais, eram muito significativos.

Embora o 8088 possa, em princípio, executar qualquer programa escrito para o 8086 e vice-versa, existem diferenças de desempenho entre os dois processadores. Enquanto o 8086 pode acessar dois bytes da memória de uma só vez, o 8088 precisa fazer dois acessos para conseguir o mesmo resultado. Assim, programas que façam amplo uso da transferência de words de e para a memória serão mais rápidos no 8086 do que no 8088.

É importante saber também que, devido à forma de conexão dos circuitos de memória ao seu barramento, o 8086 consegue acessar um word de uma só vez apenas se este iniciar num endereço par. Caso contrário, o processador será obrigado a executar dois ciclos, combinando os dados de cada um de modo a formar o word desejado. Este processo é automático e não requer esforço de programação adicional por parte do usuário, mas é importante estar ciente de que o desempenho do 8086 é melhor quando os words – sobretudo a pilha! – iniciam em endereços pares. Este posicionamento dos words recebe, em inglês, o nome de *word alignment*.

6.3 Um pequeno problema

Os efeitos da ampliação do espaço de endereçamento na arquitetura de 16 bits podem ser compreendidos a partir da análise da largura necessária para o barramento de endereços. Considerando que $1\text{MB} = 2^{20}$ bytes, vê-se que esse barramento deve ter 20 linhas, e que, portanto, os endereços que o processador deve fornecer à memória são números de 20 bits.

É claro que, durante o processamento do programa, surgem diversas oportunidades em que o processador precisa armazenar endereços de memória, e daí surge um problema: como armazenar endereços de 20 bits, se a arquitetura do processador prevê registradores de 16 bits? Como primeira tentativa, poder-se-ia pensar em incluir alguns registradores especiais de 20 bits, apenas para conter endereços. Contudo, esta idéia logo se mostra pouco prática, pois os endereços não existem apenas para serem colocados no barramento. Muitos programas fazem cálculos com endereços, o que significa que é necessário poder manipulá-los de forma semelhante a outros dados. Como consequência, os demais registradores do processador também teriam que ser ampliados para 20 bits. Mas isso teria sido um choque demasiado grande para a cultura já estabelecida, que havia firmado o byte de 8 bits como um padrão. Encarar os novos registradores de 16 bits como se fossem

registradores de 2 bytes parecia razoável, mas com 20 bits não teria sido possível chegar a um resultado que pudesse ser chamado de compatível com a tecnologia anterior. Isso poderia significar a não aceitação do produto, apesar dos muitos triunfos que o marketing já conseguiu sobre a Ciência.

Os projetistas se viram então diante de uma tarefa difícil: endereçar 1 MB de memória com registradores de 16 bits. A solução encontrada, que marcaria não só o 8086 mas, em maior ou menor grau, todos os seus sucessores até hoje, chama-se *segmentação*.

6.4 Segmentação

O princípio da segmentação consiste em combinar dois registradores de 16 bits para gerar um endereço de memória de 20 bits. O processador utiliza dois tipos diferentes de registradores nessa combinação: um *registrador de segmento* e um *registrador de offset*, e determina o endereço físico utilizando a relação:

$$\text{Endereço físico} = \text{reg. de segmento} * 16 + \text{reg. de offset.} \quad (6.1)$$

Ao observar a expressão acima, é importante perceber que a multiplicação por 16 equivale a acrescentar 4 zeros ao número binário contido no registrador de segmento, e que portanto este é transformado num número de 20 bits. É claro que esta operação não é suficiente para gerar todos os números possíveis de 20 bits, pois até aqui os 4 últimos bits são iguais a zero. A soma com o registrador de offset, porém, permite obter qualquer número de 20 bits.

É igualmente importante ter em mente que o registrador de offset é um registrador de 16 bits e que, portanto, o segundo termo da soma na equação (6.1) é um número contido no intervalo $[0, 2^{16} - 1]$, o que corresponde a uma faixa de 64 kB.

Logo, uma vez que se escolha um valor para o registrador de segmento, digamos, s , os endereços de 20 bits que podem ser gerados com o mecanismo governado por (6.1) cobrem a faixa que vai de $16s$ até $16s + 2^{16} - 1$ bytes. Diz-se, por isso, que cada número colocado num registrador de segmento representa um segmento, e que cada segmento tem 64 kB.

A compreensão do mecanismo descrito acima é fundamental para o entendimento do restante do material sobre o 8086. A figura 6.1, encontrada com variações em praticamente qualquer texto sobre o assunto, ilustra o que foi dito até aqui.

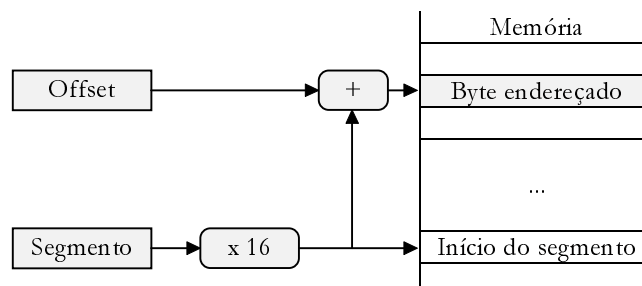


Fig. 6.1 – O mecanismo de segmentação

A título de exemplo, suponha que o registrador de segmento da figura contenha o valor 1000H e que o registrador de offset utilizado esteja com o valor 2000H. Então o segmento endereçado começa em 10000H e a posição de memória endereçada será a de endereço físico 12000H.

Uma vez fixado um valor para o registrador de segmento, o processador pode acessar os 64kB correspondentes variando apenas o conteúdo do registrador de offset. No caso do exemplo anterior, o processador pode atingir qualquer byte situado de 10000H (offset = 0) até 1FFFFH (offset = FFFFH). Esta propriedade se mostra bastante útil, uma vez que a maioria das sub-rotinas chamadas e também a maioria das estruturas de dados utilizadas por programas típicos cabe num espaço menor do que 64 kB. Desta forma, é relativamente raro que um programa precise modificar os valores dos registradores de segmento durante sua execução, o que contribui para um melhor desempenho do processador.

6.4.1 Notação

Em geral, não é necessário que o programador se preocupe com o valor do endereço físico de uma posição de memória, mas apenas com os valores dos registradores de segmento e offset utilizados para acessá-la. Existe uma convenção bastante difundida para denotar endereços definidos pelo par de registradores adotados na segmentação, que será adotada aqui. Os valores dos registradores são escritos na forma *segmento:offset*, como por exemplo 1000H:1234H, o que corresponde ao endereço físico 11234H.

6.4.2 Multiplicidade de endereços

Um aspecto da segmentação que merece atenção especial é a multiplicidade de endereços de uma mesma posição de memória. Esta decorre diretamente da expressão (6.1). A multiplicação do conteúdo do registrador de segmento por 16 faz com que cada segmento inicie numa posição de memória cujo endereço é múltiplo desse número. Assim, a menor distância possível entre dois segmentos consecutivos é de 16 bytes, como por exemplo entre os segmentos que começam em 10000H e 10010H e que correspondem a valores de registrador de segmento iguais a 1000H e 1001H, respectivamente.

Uma vez que cada offset pode atingir 64 kB dentro de um segmento, torna-se visível que existe uma superposição dos espaços de endereçamento. No caso dos dois segmentos citados, o 17.º byte do primeiro corresponde ao 1.º byte do segundo, e esta superposição se estende até o final do primeiro segmento. A verificação é simples, bastando calcular os endereços físicos:

- 17.º byte do primeiro segmento: $1000H:0010H = 1000H * 10H + 10H = 10010H$;
- 1.º byte do segundo segmento: $1001H:0000H = 1001H * 10H + 00H = 10010H$.

Desta forma, uma mesma posição de memória pode ser endereçada por diversos pares segmento:offset. Isto, em geral, não causa problemas, mas é preciso estar atento para este fato, principalmente quando se precisa comparar endereços. O simples fato de dois pares segmento:offset serem diferentes entre si não garante que a posição de memória por eles endereçada não seja a mesma.

Um bom exercício para familiarização com o mecanismo de segmentação consiste em determinar quantas maneiras diferentes existem de endereçar cada posição de memória do espaço de 1MB alcançado pelo 8086. A resposta é 4096.

6.5 Os registradores de segmento

Num programa para o 8086, distinguem-se três tipos diferentes de segmentos: de dados, de código e de pilha.

O 8086 tem quatro registradores de segmento, um para endereçar segmentos de código (CS), outro para endereçar segmentos de pilha (SS) e outros dois para endereçar segmentos de dados (DS e ES).

As instruções que o processador executa são sempre lidas a partir do segmento endereçado pelo registrador CS – não há outra opção. De forma análoga, instruções que afetam a pilha utilizam sempre o registrador SS. O acesso a dados geralmente é feito através do registrador DS, mas é possível também utilizar ES e mesmo CS e SS, se houver necessidade de endereçar dados nesses segmentos.

A tabela 6.1 apresenta os registradores de segmento do 8086.

Símbolo	Nome	Utilização mais comum
CS	Code segment	Endereçamento de código executável
DS	Data segment	Acesso a dados
ES	Extra segment	Acesso a dados
SS	Stack segment	Acesso à pilha

Tab. 6.1 – Os registradores de segmento do 8086

6.6 A visão do processador

Dados os quatro registradores de segmento, pode-se compreender a visão que o processador tem da memória. Uma vez que o acesso à memória externa sempre exige a participação de um registrador de segmento e que cada segmento abrange 64 kB, torna-se claro que o processador só “enxerga” a memória através de quatro janelas desse tamanho, e nunca está pronto para acessar todas as posições da memória diretamente. A figura 6.2 ilustra uma situação hipotética em que os quatro registradores de segmento apontam para diferentes regiões da memória.

Nada impede, porém, que haja superposição parcial ou mesmo total dos segmentos. O processador não impõe qualquer tipo de restrição aos valores que podem ser colocados nos registradores de segmento, de modo que o programador pode distribuí-los conforme lhe convier. A superposição de segmentos é comum em programas pequenos, pois nem sempre são necessários todos os 64 kB disponíveis em cada segmento. No entanto, isto torna possível que um offset errado a partir de um segmento venha a invadir outro, danificando dados lá contidos. Por exemplo, se existir superposição dos segmentos de código e de dados, um erro no offset dentro do segmento de dados pode corromper o código do programa. A responsabilidade por este tipo de erro cabe exclusivamente ao programador.

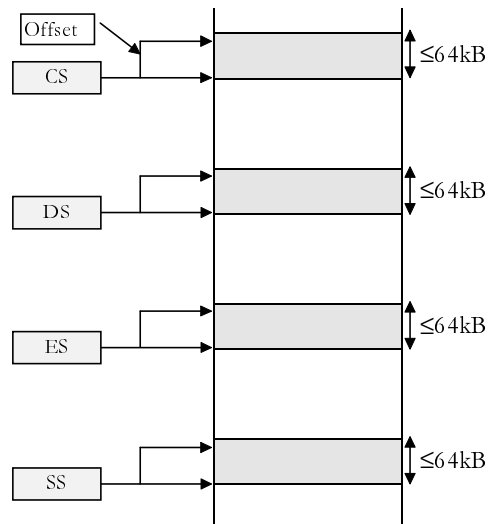


Fig. 6.2 – Endereçamento de memória com 8086

6.7 O modelo de programação

O usuário vê o 8086 de acordo com o modelo apresentado na figura 6.3, que apresenta os registradores do processador.

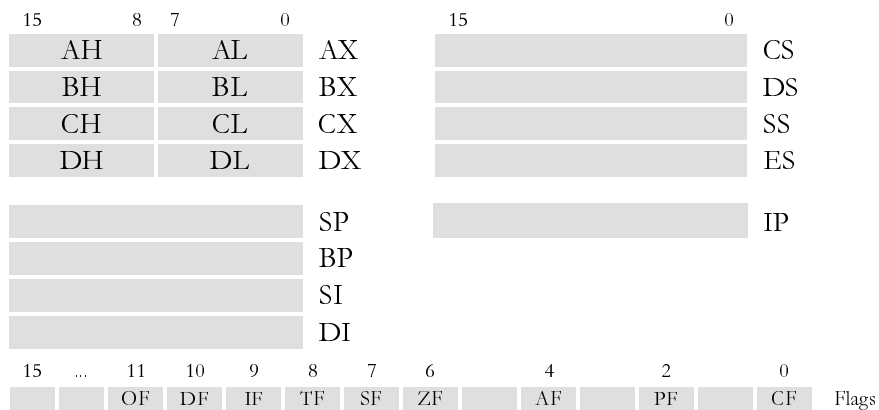


Fig. 6.3 – O modelo de programação do 8086

São ao todo 14 registradores de 16 bits, incluindo os já mencionados registradores de segmento, CS, DS, SS e ES. Além desses, aparecem os assim

chamados registradores de uso geral, AX – DI, o ponteiro de instruções, IP (instruction pointer, que faz o papel do PC do 8085) e os flags.

Os registradores AX – DX podem ter cada um de seus bytes endereçados separadamente. Para isto, as instruções utilizam os nomes AH – DH para os bytes mais significativos e os nomes AL – DL para os bytes menos significativos desses registradores. Embora sejam classificados pela Intel como registradores de uso geral, esses registradores não são completamente equivalentes entre si. Por exemplo, as operações mais complexas requerem, em geral, que se use o registrador AX, que reflete algumas das características do acumulador do seu antecessor, o 8085. O registrador DX é necessário nas operações de multiplicação e divisão de 32 bits e em algumas operações de I/O, e o registrador CX tem o papel de contador nas instruções que dão suporte à construção de loops.

Os outros registradores de uso geral, SP – DI, são registradores de endereço e por isso podem ser acessados apenas como registradores de 16 bits. Os registrador SP é o ponteiro da pilha; BP pode servir como registrador de uso geral mas desempenha também um papel importante relacionado à pilha, na passagem de parâmetros e na criação de variáveis locais (v. capítulo 7). SI e DI têm aplicações de uso geral e são também empregados pelas instruções de manipulação de strings.

6.8 Considerações sobre a linguagem ASM-86

A linguagem assembly desenvolvida pela Intel para o 8086 chama-se ASM-86. Detalhar todas as características desta linguagem está além do objetivo deste texto, mas existem algumas características que são essenciais para a compreensão dos programas que serão estudados, e que por isso merecem atenção especial.

O conceito mais importante e diferente de outras linguagens assembly é o conceito de *tipo*. Cada símbolo de um programa ASM-86, seja o nome de uma variável, um sinônimo para um endereço ou um sinônimo para uma constante, tem um determinado tipo. Muitas vezes, o assembler não precisa que o programador diga explicitamente qual o tipo de um símbolo, pois pode deduzi-lo por conta própria. Por exemplo, em

```
MOV AX,[BX] ; copia o word endereçado por BX para AX
```

o assembler pode deduzir que o conteúdo a ser copiado da memória é um word, pois o registrador AX é de 16 bits. Mas no caso da instrução

```
INC [BX] ; incrementar byte ou word ???
```


o assembler não tem como saber se deve incrementar o byte que reside no offset BX ou se deve incrementar o word que reside em BX, BX + 1. Neste caso, o assembler geraria uma mensagem de erro, obrigando o programador a decidir qual dos dois efeitos deseja que a instrução tenha. O programador tem que dizer qual o tipo da variável endereçada indiretamente por BX, escrevendo

```
INC BYTE PTR [BX] ; incrementa o byte de offset BX
```

para incrementar o byte cujo offset é o valor contido em BX, ou

```
INC WORD PTR [BX] ; incrementa o word de offset BX
```

para incrementar o word que se encontra nos endereços BX e BX + 1.

Os tipos definidos na linguagem ASM-86 são os seguintes:

- BYTE PTR: referencia uma variável de 1 byte;
- WORD PTR: referencia uma variável de 1 word;
- DWORD PTR: referencia uma variável de 2 words;
- NEAR PTR: referencia o endereço de destino de uma instrução de desvio do tipo *near*;
- FAR PTR: idem, para desvios do tipo *far*;
- NUMBER: o valor do símbolo com este tipo, que raramente precisa ser utilizado, é um sinônimo para uma constante de 16 bits.

6.9 Formato das instruções do 8086

O formato escolhido pelos projetistas para as instruções de um microprocessador é uma solução de compromisso entre elegância, ocupação de memória e desempenho. Por exemplo, para privilegiar a estética, poder-se-ia reservar um pedaço de cada instrução para conter o código que a identifica (assim como acontece com o primeiro byte das instruções do 8085). No entanto, poderia haver instruções que não necessitassem de todo este espaço, e com isso haveria desperdício em alguns casos. Este desperdício afetaria negativamente a economia de memória e o desempenho.

No caso do 8086, o que se fez foi procurar aproveitar cada bit disponível, de modo a compactar ao máximo as instruções e otimizar a utilização da memória e a velocidade de leitura das instruções. Em contrapartida, sua codificação é muito mais complexa do que no 8085, pois agora o código de uma instrução não precisa ocupar exatamente 1 byte e os bits não ocupados são utilizados com outras finalidades. O comprimento das instruções pode variar de 1 a 6 bytes.

A título de exemplo, a figura 6.4 traz o formato das instruções de movimentação de dados imediatos para memória ou entre registradores. O código da instrução ocupa os 6 bits mais significativos do primeiro byte, enquanto os bits 1 e 0 influenciam o restante da instrução. O bit *w*, por exemplo, serve para distinguir as instruções que manipulam bytes (*w* = 0) das instruções análogas que manipulam words (*w* = 1).

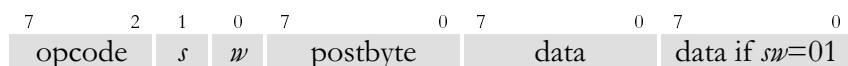


Fig. 6.4 – Exemplo de formato de instrução do 8086

O postbyte é dividido em três campos, conforme a figura 6.5. Se o campo *mod* contém o valor binário 11, então o operando da instrução é um registrador, identificado por um código de 3 bits colocado no campo *r/m*. Instruções que referenciam dois registradores podem utilizar o campo *reg* para identificar o segundo registrador. Outros valores colocados no campo *mod* significam que este deve ser decodificado em conjunto com *r/m* para determinar o modo de endereçamento a ser utilizado.



Fig. 6.5 – Detalhamento do postbyte

O papel do exemplo acima é apenas o de proporcionar ao leitor uma idéia de como a codificação é feita. É importante notar como se consegue aproveitar os bits disponíveis, mas não se preocupar com a memorização de detalhes, que são exclusivos do 8086 e portanto de pouca utilidade para a assimilação dos conceitos considerados importantes neste texto.

6.10 Modos de endereçamento

Esta seção descreve os diversos modos de endereçamento do 8086. Embora tratem especificamente deste componente, as informações apresentadas são de utilidade geral, uma vez que muitos modos de endereçamento encontrados em outros microprocessadores e microcontroladores são semelhantes aos descritos aqui.

Os modos de endereçamento são as diferentes maneiras que as instruções do processador têm para especificar a localização de seus operandos. Um operando pode fazer parte de uma instrução, ou então pode estar localizado em um registrador ou na memória. Dependendo do caso, diferentes modos são

necessários. Por exemplo, para inicializar um registrador com um valor fixo e conhecido em tempo de montagem¹, em geral será conveniente utilizar o modo imediato, que aceita uma constante; para varrer uma tabela em busca de alguma informação, será necessário algum tipo de endereçamento que permita variar o índice do elemento endereçado em tempo de execução².

6.10.1 Endereçamento via registrador

Neste modo de endereçamento, o operando a ser utilizado se encontra num registrador, referenciado pela instrução. Podem ser utilizados os registradores AX-DI ou ainda os registradores de 8 bits, AH-DL.

Por exemplo, a instrução

```
MOV AX,BX           ; copia o conteúdo de BX para AX
```

faz referência ao registrador BX, porque é lá que está contido o operando a ser copiado para o registrador AX.

Da mesma forma, em

```
CMP AL,DL           ; seta os flags de acordo com o  
                    ; resultado da subtração AL-DL
```

o dado a ser comparado com o valor de AL está localizado em DL.

6.10.2 Endereçamento imediato

Aqui, o operando faz parte da própria instrução. Este modo é utilizado para atribuir valores constantes de 8 ou de 16 bits a registradores ou posições de memória, como em

```
MOV AX,1000H        ; AX = 1000H
```

ou

```
CMP SI,0000H        ; seta os flags de acordo com o  
                    ; resultado da subtração SI-0000H.
```

O modo imediato do 8086 tem duas pequenas limitações: não está disponível para atribuição de valores aos registradores de segmento nem para a instrução PUSH.

Pode-se perguntar agora se as instruções utilizadas nos exemplos não utilizam também o modo via registrador, uma vez que há registradores especificados.

¹ Tempo de montagem: refere-se à etapa de desenvolvimento do programa. Valores conhecidos em tempo de montagem são constantes determinadas de antemão pelo programador ou calculadas pelo assembler, e não podem ser modificados durante a execução.

² Tempo de execução: refere-se ao programa em execução.

Isto é verdade, e a presença de dois modos de endereçamento pode acontecer em todas as instruções de dois operandos. Aqui é claro que o modo imediato se refere ao operando de origem. Em outros pontos do texto, o operando de que se trata será citado explicitamente, se houver possibilidade de confusão.

6.10.3 Endereçamento absoluto ou direto

Neste modo, o operando que se deseja endereçar reside na memória, e a instrução contém o endereço do operando. A figura 6.6 ilustra o funcionamento da instrução

```
MOV AX,[1000H]      ; copia para AX o word localizado
                    ; nos offsets 1000H e 1001H.
```

Endereço	Dado	
1000H	34H	→ AX 1234H
1001H	12H	
1002H	25H	
...	...	

Fig. 6.6 – Endereçamento absoluto: MOV AX,[1000H]

Existe também um modo de endereçamento absoluto longo, utilizado apenas em instruções de desvio do tipo far, onde são fornecidos segmento e offset do destino do desvio:

```
JMP 1234H:5678H; salto incondicional para 1234H:5678H
```

6.10.4 Endereçamento indireto

Tal como no caso anterior, o operando reside na memória. A instrução, porém, não especifica o offset do operando, mas um registrador de 16 bits, que contém esse offset. Podem ser utilizados os registradores BX, BP, DI e SI.

A figura 6.7 ilustra o funcionamento da instrução

```
MOV AX,[BX]          ; AX recebe o word de offset BX.
```

De forma análoga, a instrução

```
INC BYTE PTR [DI]
```

incrementa o byte cujo offset está contido em DI. Note a utilização explícita do tipo BYTE PTR neste caso, necessário para que o assembler possa decidir se o programa deve incrementar o byte que reside no endereço DI ou o word que reside em DI, DI+1. No exemplo anterior, o assembler pode deduzir que

o operando de origem é do tipo WORD PTR, porque o operando de destino (AX) é um registrador de 16 bits. Por isso, naquele caso o tipo não precisa ser explicitado.

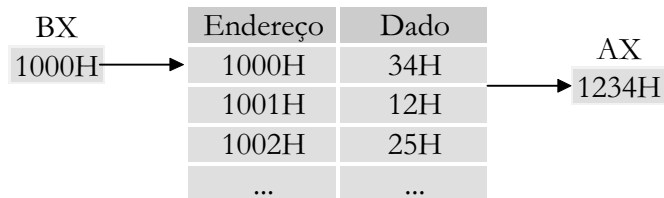


Fig. 6.7 – Endereçamento indireto: MOV AX,[BX]

O endereçamento indireto é muito utilizado para trabalhar com tabelas. O trecho de código a seguir, por exemplo, varre uma tabela de N bytes cujo offset inicial é dado pelo símbolo TABELA. O loop de varredura se encerra assim que for encontrado um elemento diferente de zero ou quando o final da tabela é atingido sem que a busca tenha tido sucesso.

```

        LEA BX,TABELA          ; BX = offset inicial
        MOV CX,N               ; CX = tamanho da tabela
LOOP:   CMP BYTE PTR [BX],00H
        JNZ ACHEI
        INC BX
        DEC CX
        JNZ LOOP
        ...                    ; busca terminou sem sucesso
ACHEI:  ...                    ; busca terminou com sucesso

```

No caso em que a busca termina com sucesso, BX aponta para o primeiro byte não nulo da tabela.

6.10.5 Endereçamento indexado

A instrução especifica uma constante, denominada base, e um registrador de índice, cujo conteúdo é somado à base para formar o endereço do operando. Podem ser utilizados os registradores BP, BX, SI e DI.

A figura 6.8 ilustra o funcionamento da instrução

```
MOV AX,0100H[DI]      ; compara AX com word em 100+DI,
```

que também pode ser escrita como

```
MOV AX,[0100H+DI]    ; idem, outra notação.
```

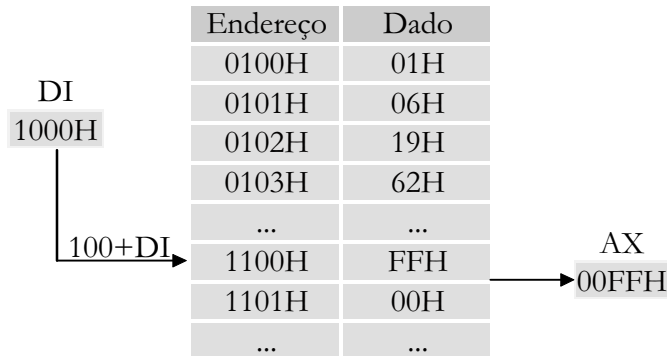


Fig. 6.8 – Endereçamento indexado: MOV AX,[100H+DI]

O endereçamento indexado permite percorrer uma tabela utilizando um offset local, isto é, um offset que vale 0 para o primeiro byte da tabela, 1 para o segundo byte e assim por diante. É útil para acessar elementos de tabelas, quando a posição inicial da tabela (a base) é conhecida em tempo de montagem, mas o offset do elemento a acessar varia em tempo de execução. O exemplo da seção 6.10.4 pode ser reescrito da seguinte forma:

```

MOV BX,0000H          ; BX = offset local inicial
LOOP:  CMP BYTE PTR [TABELA + BX],00H
       JNZ ACHEI
       INC BX
       CMP BX,N
       JNZ LOOP
       ...              ; busca terminou sem sucesso
ACHEI:  ...              ; busca terminou com sucesso

```

No caso em que a busca termina com sucesso, BX contém o offset local do primeiro byte não nulo da tabela.

6.10.6 Endereçamento baseado

Este modo também utiliza uma base e um deslocamento, mas, ao contrário do que acontece no endereçamento indexado, aqui a base é variável e o índice é constante. Por isso, a instrução especifica um registrador de base e uma constante, que serve como deslocamento em relação a essa base. O endereço do operando é dado pela soma da base com a constante. Podem ser utilizados como base os registradores BP, BX, SI e DI, embora normalmente se utilizem BX e BP. A figura 6.9 ilustra o funcionamento da instrução

```
MOV [BX+0005H],CX      ; copia CX para o offset BX+5
```

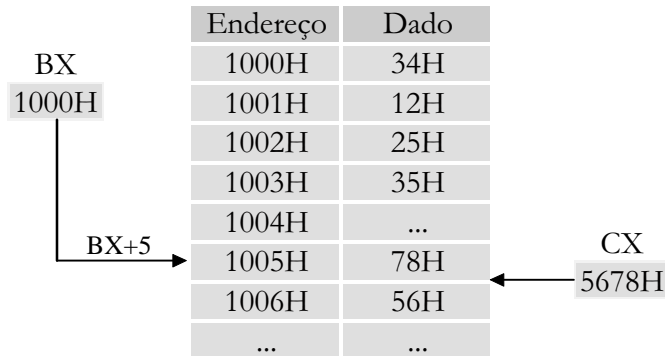


Fig. 6.9 – Endereçamento baseado: MOV [BX+0005H],CX

O endereçamento baseado é útil para acessar elementos de tabelas cujo offset local é conhecido em tempo de montagem, enquanto a posição inicial da tabela varia em tempo de execução.

A título de exemplo, pode-se imaginar um programa em que existam várias tabelas de mesma estrutura, contendo dados como o nome e a idade dos alunos de uma sala de aula, e que a idade de cada um se encontre sempre no offset 5 da tabela que lhe corresponde. As tabelas têm todas o mesmo tamanho, dado pela constante SIZE, e ficam dispostas uma após a outra, sem elementos estranhos entre elas. O programa poderia então determinar quem é o aluno mais velho, varrendo o conjunto de tabelas. Para isto, colocaria em BX o offset inicial de cada uma e leria de BX+5 a idade de cada um:

```

        LEA BX,ALUNOS      ; offset da primeira tabela
        MOV DI,BX          ; inicializa aluno mais velho
        MOV AL,00H         ; valor inicial da idade
        MOV CX,N           ; número de alunos
LOOP:   CMP AL,[BX+5]       ; compara maior idade encontrada
        JB OK              ; até agora com o valor em [BX+5]
        MOV AL,[BX+5]      ; AL contém a idade mais alta
        MOV DI,BX          ; DI aponta para o aluno mais velho
OK:     ADD BX,SIZE         ; BX -> tabela seguinte
        DEC CX
        JNZ LOOP

```

Ao final do loop, DI aponta para o aluno mais velho da classe (ou para primeiro aluno da lista que tem a idade mais alta, caso haja mais alunos com a mesma idade).

O endereçamento baseado dá ainda suporte a algumas aplicações importantes da pilha, estudadas no capítulo 7.

6.10.7 Endereçamento baseado indexado

Este modo é uma combinação dos dois anteriores. A instrução especifica registradores tanto para a base quanto para o índice; o endereço efetivo do operando é formado pela soma de ambos e, se o programador desejar, de mais uma constante opcional. Os registradores de base podem ser BP ou BX e os registradores de índice podem ser SI ou DI.

A figura 6.10 ilustra o funcionamento da instrução

```
MOV AX,[BX+SI] ; AX recebe word em BX + SI
```

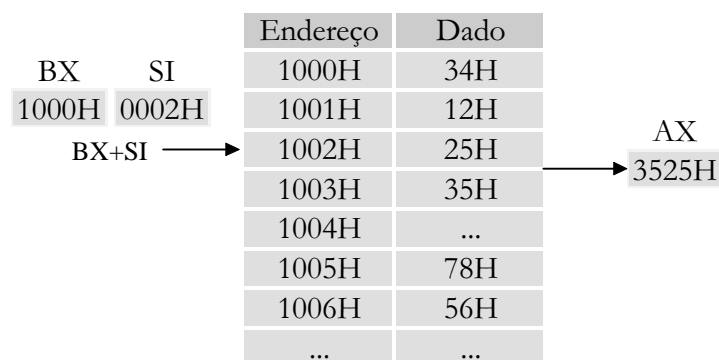


Fig. 6.10 – Endereçamento baseado indexado: MOV AX,[BX+SI]

O uso da constante opcional é ilustrado pela instrução

```
CMP CL,[100H+BX+DI] ; seta flags de acordo com o  
                      ; resultado subtração CL-byte
```

O modo baseado indexado é utilizado para acessar tabelas onde tanto a base quanto o índice variam em tempo de execução. Um exemplo de aplicação pode ser encontrado no capítulo 19, que o emprega para trabalhar com estruturas matriciais.

6.10.8 Endereçamento relativo

Este modo de endereçamento é utilizado nas instruções JMP e CALL dos tipos near e short e em todos os desvios condicionais. Para facilitar a compreensão deste modo, é interessante lembrar primeiro como se dá a codificação de uma instrução de desvio no caso do microprocessador 8085. Por exemplo, considere a instrução JC 2015H, codificada no endereço 2010H de um programa:

```
2010H DA 15 20 JC 2015H
```


Aqui, DAH é o opcode da instrução JC e o endereço de destino, 2015H, é determinado pelo processador através da leitura dos dois bytes seguintes da instrução. Como a instrução contém um endereço, este é um caso de endereçamento absoluto.

No caso do 8086, porém, a codificação é bem diferente, conforme ilustra o trecho de código abaixo, que subtrai DX de AX e limita o resultado a zero, caso não seja positivo:

```
1CD7:0100 29 D0          SUB AX,DX
1CD7:0102 73 03          JNC 0107
1CD7:0104 B8 00 00       MOV AX,0000
1CD7:0107 ...
```

Note agora a codificação da instrução JNC: o opcode, 73H, é seguido apenas do número 03H e o offset de destino, 107H, não aparece na codificação. Em vez disso, o segundo byte da instrução é utilizado pelo processador para determinar o endereço de destino. A execução da instrução JNZ consiste em somar o valor 03H ao conteúdo do registrador IP. No instante em que a instrução é propriamente executada, IP já foi atualizado para apontar para a instrução seguinte, de modo que a conta que o processador faz é:

Offset de destino = offset da instrução seguinte + deslocamento

Offset de destino = 0104H + 0003H = 0107H.

As seguintes observações são importantes: o exemplo ilustra uma instrução de desvio do tipo short, que utiliza apenas um byte para indicar o endereço de destino. Isto significa que o offset de destino tem que estar, no máximo, 128 bytes antes ou 127 bytes depois do offset da instrução seguinte, porque esses são, respectivamente, o menor e o maior número com sinal que podem ser representados com 1 byte. Se o endereço de destino cair fora desta faixa, o assembler escolherá o modo near, que usa deslocamentos de 16 bits e permite, com isso, atingir qualquer ponto dentro do segmento. No entanto, apenas os desvios incondicionais podem ser do tipo near, e por isso pode ser necessário combinar um desvio condicional com um desvio incondicional, caso o destino fique fora da faixa alcançável pelos desvios do tipo short.

Além disso, convém saber que, no caso dos desvios do tipo short, o processador transforma automaticamente o número a ser somado ao registrador IP em um número de 16 bits, de acordo com o seu sinal. Números positivos são estendidos antepondo-se zeros (03 vira 0003) e números negativos são estendidos antepondo-se 1's (F4H vira FFF4H). Isto permite que o processador some sempre o valor correto a IP para encontrar o endereço de destino.

Em qualquer caso, o valor contido na instrução não é o endereço de destino, mas sim a diferença, em bytes, entre o valor atual do registrador IP e o valor que este deverá assumir para realizar o desvio. Desta forma, o endereço de destino não é mais absoluto, mas sim dado em relação a IP, e por isso se chama *endereço relativo*.

A grande vantagem do endereçamento relativo aparece quando trechos de código como os mostrados nos exemplos precisam ser mudados de lugar na memória durante a execução do programa, tarefa que faz parte do gerenciamento de memória dos sistemas operacionais mais sofisticados.

Se fosse necessário mover o programa dado como exemplo para o 8085 para um endereço 100H bytes adiante, a instrução JC 2015H teria que ser recodificada para JC 2115H, valendo o mesmo para todas as instruções de desvio presentes no programa. No caso do 8086, a utilização do endereçamento relativo permite que o código possa ser movido sem qualquer alteração, pois a diferença entre o destino e a origem do desvio não se altera quando se muda o programa de lugar.

6.10.9 Determinação do segmento utilizado

Diversos modos de endereçamento estudados nas seções anteriores envolvem o uso de endereços de 16 bits para localizar um operando. Em todos esses casos, o endereço em questão é o offset da posição de memória onde o operando se encontra, mas é claro que a localização definitiva de qualquer posição só pode ser feita se for especificado, de alguma forma, o registrador de segmento que deve ser utilizado em conjunto com esse offset.

Na maioria dos casos, a especificação do registrador de segmento é feita de forma implícita, pois para cada instrução existe um registrador default. A tabela 6.2 apresenta esses registradores de acordo com o tipo de operação realizada, bem como as alternativas possíveis quando se deseja mudar esse default; aparecem ainda os registradores que podem ser utilizados como offsets.

Operação	Default	Alternativas	Offset
Busca de instruções	CS	Nenhuma	IP
Acesso à pilha	SS	Nenhuma	SP
Movimentação de dados, exceto com BP	DS	CS, ES, SS	Reg de uso geral / endereço absoluto
Movimentação de dados com BP (end. baseado)	SS	CS, ES, DS	BP ou BP + constante
Origem de dados na manipulação de strings	DS	CS, ES, SS	SI
Destino de dados na manipulação de strings	ES	Nenhuma	DI

Tab. 6.2 – Segmentos default e alternativas

Resta ainda explicar como se especifica um segmento alternativo, quando se deseja mudar o default. Esta mudança é feita mediante utilização do chamado *segment override prefix* ou prefixo de modificação de segmento na instrução em questão. Este prefixo é constituído do nome do segmento seguido por um sinal de ‘:’. Por exemplo, a instrução

```
MOV AX, ES:BX
```

copiar para AX o word cujo offset está em BX, mas de dentro do segmento ES, e não mais do segmento DS, que seria utilizado na ausência do prefixo ES:.

Em alguns casos, o uso dos prefixos de modificação de segmento pode ser automatizado com a diretiva ASSUME, conforme explica o anexo 6.

7 Parâmetros e variáveis locais

7.1 Introdução

Como vimos em capítulos anteriores, a pilha tem um papel importante no mecanismo de chamada de sub-rotinas e de tratadores de interrupção, assim como no armazenamento temporário de valores contidos nos registradores do processador.

Este capítulo trata de mais dois usos importantes da pilha: a passagem de parâmetros para sub-rotinas e a criação de variáveis locais.

7.2 Passagem de parâmetros para sub-rotinas

É comum que uma sub-rotina aceite um ou mais parâmetros, os quais utiliza para realizar seu processamento. Por exemplo, uma sub-rotina capaz de comparar duas strings precisa saber onde se encontram estas strings para fazer a comparação, e uma sub-rotina que calcule uma função $f(x)$ precisa ser informada do valor de x para poder fazer o cálculo. Os valores necessitados pelas sub-rotinas devem ser fornecidos pela parte do programa que as chama, operação esta conhecida como *passagem de parâmetros*.

A passagem de parâmetros pode ser feita de diversas formas. São apresentadas a seguir três formas de passagem de parâmetros, as duas primeiras servindo principalmente para comparação com a terceira, que é o principal objeto deste estudo.

7.2.1 Passagem através de registradores

A forma mais simples é a *passagem de parâmetros através de registradores*. Neste caso, os valores dos parâmetros são colocados pelo programa chamador em alguns dos registradores do processador, de acordo com uma convenção estabelecida

pelo programador da sub-rotina. Por exemplo, uma sub-rotina que imprima uma mensagem na tela pode considerar que o registrador DX contém o offset desta mensagem no segmento de dados atual. Cabe então ao programa chamador colocar este offset no registrador DX, antes de efetuar a chamada para a sub-rotina. A listagem a seguir ilustra o emprego deste mecanismo.

```

CODIGO  SEGMENT
...
INICIO: MOV AX,DADOS
        MOV DS,AX
        LEA DX,MSG1           ; DX CONTÉM OFFSET DE MSG1
        CALL SHOW
        LEA DX,MSG2           ; DX CONTÉM OFFSET DE MSG2
        CALL SHOW
...
SHOW    PROC NEAR              ; APRESENTA MSG DE OFFSET DS:DX
        MOV AH,09H            ; WRITE STRING
        INT 21H
        RET
SHOW    ENDP
CODIGO  ENDS

DADOS   SEGMENT
MSG1    DB 'MENSAGEM 1$'
MSG2    DB 'MENSAGEM 2$'
DADOS   ENDS

```

7.2.2 Passagem através de variáveis globais

Este mecanismo é semelhante ao da passagem através de registradores. A diferença está apenas no fato de se utilizarem variáveis ao invés de registradores do processador para passar os parâmetros. O programa chamador deve colocar os parâmetros nestas variáveis antes da chamada da sub-rotina, que irá buscá-los para efetuar seu processamento. A listagem seguinte mostra como ficaria o programa do exemplo anterior utilizando a variável global S_PTR (string pointer), definida no segmento de dados.

```

CODIGO  SEGMENT
...
INICIO: MOV AX,DADOS
        MOV DS,AX
        LEA S_PTR,MSG1        ; ENDEREÇO DA MSG1 EM S_PTR
        CALL SHOW
        LEA S_PTR,MSG2        ; ENDEREÇO DA MSG2 EM S_PTR
        CALL SHOW
...

```

```
SHOW    PROC NEAR                ; APRESENTA MSG EM S_PTR
        MOV AH,09H              ; WRITE STRING
        MOV DX,S_PTR            ; DX APONTA P/ MSG A APRESENTAR
        INT 21H
        RET
SHOW     ENDP
CODIGO   ENDS

DADOS    SEGMENT
MSG1     DB 'MENSAGEM 1$'
MSG2     DB 'MENSAGEM 2$'
S_PTR    DW ?
DADOS    ENDS
```

7.2.3 Passagem através da pilha

Este mecanismo consiste em colocar os parâmetros a serem passados para a sub-rotina na pilha, antes da chamada. A sub-rotina busca então os parâmetros na pilha para realizar seu processamento.

Embora o princípio de funcionamento seja simples, há alguns fatores a considerar, que, se ignorados, podem levar o programa a se perder. São eles:

- uma vez pronta a sub-rotina, a ordem em que os parâmetros devem ser passados fica fixa e deve ser respeitada pelo programador que a chama;
- para encontrar os parâmetros na pilha, a sub-rotina precisa levar em conta que, além destes, a pilha conterá também o endereço de retorno da sub-rotina;
- o programa chamador, que colocou os parâmetros na pilha, é responsável também por removê-los de lá após a chamada.

Enquanto a primeira consideração dispensa explicações adicionais, as outras duas merecem maior atenção. A figura 7.1 ilustra o estado da pilha de um programa hipotético, após a chamada de uma sub-rotina *near* que recebe dois parâmetros através da pilha.

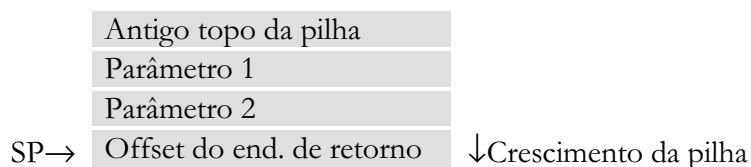


Fig. 7.1 – A pilha com dois parâmetros

Esta figura indica que a sub-rotina tem que se basear no valor de SP para ter acesso aos valores dos parâmetros recebidos. Observando-se a figura pode-se

concluir que, imediatamente após a chamada da sub-rotina, o último parâmetro passado estará no endereço SP+2, o penúltimo em SP+4, e assim por diante. No entanto, a utilização do registrador SP para endereçar os parâmetros é extremamente inconveniente, uma vez que a própria sub-rotina faz, em geral, uso da pilha. Com isso, o valor de SP variaria durante a execução da sub-rotina, o que significa, por exemplo, que o último parâmetro precisaria ser referenciado como SP+2($n+1$), onde n é o número de words colocados na pilha pela sub-rotina e ainda não retirados.

É possível evitar este transtorno através da criação de um ponteiro fixo para a região da pilha que contém os parâmetros, e que é então utilizado pela sub-rotina para endereçá-los. O registrador BP do microprocessador 8086 foi concebido com esta finalidade, e é utilizado da seguinte forma¹:

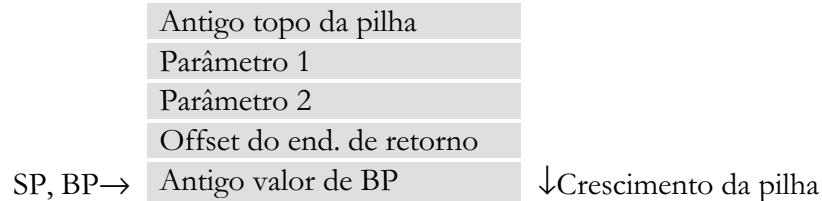
- a sub-rotina inicia salvando o valor de BP na pilha, para poder restaurá-lo antes de retornar;
- copia então o valor atual de SP para BP. Desta forma, BP vira um ponteiro fixo para a posição da pilha que contém o valor de BP quando da entrada na sub-rotina;
- o endereço BP+2 contém o offset do endereço de retorno;
- no caso de sub-rotinas do tipo *far*, o endereço BP+4 contém o segmento do endereço de retorno; no caso de sub-rotinas do tipo *near*, contém o último parâmetro passado;
- os endereços seguintes (BP+6, BP+8, ...) contêm os demais parâmetros.

Para implementar este mecanismo, as sub-rotinas iniciam sempre com a seqüência:

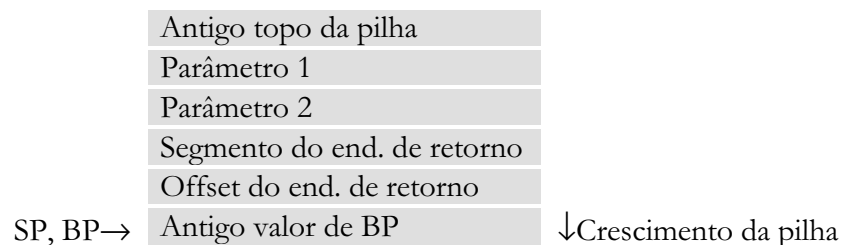
```
PUSH BP
MOV BP, SP
```

A figura 7.2 mostra a pilha para sub-rotinas do tipo *near* após a execução destas instruções. A partir deste ponto, a sub-rotina mantém o valor de BP inalterado e passa a endereçar os parâmetros 2 e 1 por BP+4 e BP+6, respectivamente. A pilha pode ser utilizada como em qualquer outro ponto do programa, pois as alterações no valor de SP não afetam mais a forma de endereçar os parâmetros.

¹ Embora seja possível imaginar diversas formas de utilizar o registrador BP para resolver o problema, a forma apresentada se destaca por ser a adotada pelos principais compiladores da linguagem C, como os da Borland e da Microsoft. Por isso, o conhecimento do material exposto aqui será útil àqueles que precisarem trabalhar com detalhes em Assembly quando envolvidos com programação de mais alto nível.

**Fig. 7.2 – A pilha após o armazenamento do valor de BP**

No caso das sub-rotinas *far*, os endereços dos parâmetros são BP+6 e BP+8, respectivamente, porque o endereço de retorno é composto de duas palavras, conforme ilustra a figura 7.3.

**Fig. 7.3 – A pilha em uma sub-rotina FAR**

Note ainda que as instruções de movimentação de dados que utilizam o registrador BP usam, por default, o registrador de segmento SS, de modo que instruções como `MOV AX,[BP+6]` não precisam de qualquer alteração para buscar o dado do segmento correto.

O final de uma sub-rotina é, tipicamente:

```
POP BP
RET
```

Convém lembrar que existem dois tipos de instruções RET, ambas utilizando o mesmo mnemônico. A primeira, denominada RET NEAR, faz com que o processador retire apenas uma palavra do topo da pilha, que considera como o offset do endereço de retorno e coloca no registrador IP. A segunda, RET FAR, faz com que sejam recuperadas duas palavras, que o processador considera como offset e segmento do endereço de retorno, nesta ordem, e coloca nos registradores IP e CS. Qualquer que seja o caso, se a rotina foi chamada corretamente, isto é, de acordo com o tipo de RET que determina seu tipo (*near* ou *far*), a situação da pilha após o término da sub-rotina será a da figura 7.4.

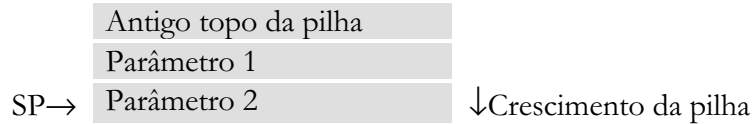


Fig. 7.4 – A pilha após o retorno da sub-rotina

O próximo ponto a tratar é a remoção dos parâmetros da pilha pelo programa chamador. Para tanto, o programa chamador poderia colocar, imediatamente após a instrução `CALL`, duas (neste caso) instruções `POP` para um registrador qualquer, o que eliminaria os parâmetros. Embora funcione, este método geralmente não é utilizado, pelos seguintes motivos:

- pode não haver nenhum registrador disponível para se fazer os `POPs`;
- o método é ineficiente quando o número de parâmetros a eliminar é muito grande.

Em lugar disso, o que se faz é simplesmente adicionar o valor $2n$ ao registrador `SP`, onde n é o número de words empilhados como parâmetros antes da chamada. O código completo da chamada de uma sub-rotina `SUB` que aceite dois parâmetros como no exemplo acima ficaria então assim:

```
MOV AX,PARAM1
PUSH AX
MOV AX,PARAM2
PUSH AX
CALL SUB
ADD SP,4                ; REMOVE OS PARÂMETROS APÓS O USO
```

7.2.4 Aninhamento de sub-rotinas (nesting)

É comum que programas sejam estruturados de forma que uma sub-rotina chame outras sub-rotinas. A análise a seguir mostra que o mecanismo de passagem de parâmetros através da pilha funciona também nestes casos, graças ao fato de que o registrador `BP` é sempre salvo na pilha no início de cada sub-rotina e restaurado antes do retorno.

Suponha um programa principal que chame uma sub-rotina *near* de nome `SUB1`, passando-lhe dois parâmetros, `P1S1` e `P2S1`:

```
MOV AX,P1S1
PUSH AX                ; PRIMEIRO PARÂMETRO NA PILHA
MOV AX,P2S1
PUSH AX                ; SEGUNDO PARÂMETRO NA PILHA
CALL SUB1
ADD SP,4                ; PARÂMETROS REMOVIDOS
```

Suponha ainda que a sub-rotina SUB1 chame uma segunda sub-rotina *far* de nome SUB2, que recebe três parâmetros, P1S2, P2S2 e P3S2:

```
SUB1    PROC NEAR
        PUSH BP
        MOV BP,SP
        ...           ; ACESSO A P1S1 E P2S1
        MOV AX,P1S2
        PUSH AX       ; PRIMEIRO PARÂMETRO NA PILHA
        MOV AX,P2S2
        PUSH AX       ; SEGUNDO PARÂMETRO NA PILHA
        MOV AX,P3S2
        PUSH AX       ; TERCEIRO PARÂMETRO NA PILHA
        CALL SUB2
        ADD SP,6       ; PARÂMETROS REMOVIDOS
        ...           ; ACESSO A P1S1 E P2S1
                        ; CONTINUA FUNCIONANDO!

        POP BP
        RET
SUB1    ENDP
```

Depois do retorno da sub-rotina SUB2, a sub-rotina SUB1 pode voltar a acessar os parâmetros P1S1 (em [BP + 6]) e P2S1 (em [BP + 4]). Isto é possível porque SUB2, embora modifique o valor de BP, restaura esse registrador antes de retornar:

```
SUB2    PROC FAR
        PUSH BP
        MOV BP,SP
        ...           ; ACESSO A P1S2, P2S2 e P3S2
        POP BP        ; BP VOLTA AO VALOR NECESSÁRIO EM SUB1
        RET
SUB2    ENDP
```

A situação da pilha imediatamente antes de a sub-rotina SUB1 começar a empilhar os parâmetros para a chamada de SUB2 é a mesma da figura 7.2. A figura 7.5 mostra como fica a pilha dentro da sub-rotina SUB2, depois que esta executa as duas primeiras instruções, posicionando BP para acessar seus parâmetros.

Depois da instrução POP BP no final de SUB2, BP volta a ter o valor que lhe tinha sido atribuído em SUB1. Desta forma, a sub-rotina SUB1 pode voltar a acessar seus parâmetros da mesma forma como vinha fazendo antes de chamar SUB2. Generalizando, enquanto houver espaço na pilha, o mecanismo de passagem de parâmetros continua funcionando, independentemente do número de sub-rotinas aninhadas.

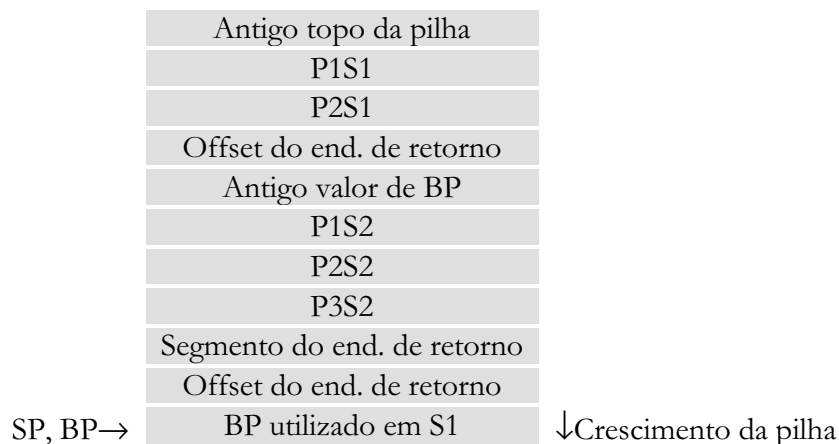


Fig. 7.5 – Passagem de parâmetros em sub-rotinas aninhadas

7.2.5 Comparação entre os métodos

A principal vantagem do método de passagem por registradores é a velocidade, uma vez não exige acesso à memória externa. No entanto, sofre das seguintes desvantagens:

- nem sempre há registradores livres em número suficiente para acomodar todos os parâmetros;
- é difícil estabelecer uma convenção sobre em que ordem os registradores devem ser utilizados, à medida que aumenta o número de parâmetros;
- em consequência, aumentam a complexidade da documentação da sub-rotina e o risco de se cometerem erros na sua chamada.

A passagem por variáveis globais elimina a primeira das desvantagens acima, mas perde em velocidade. Além disso, sofre de problemas semelhantes aos dois últimos, pois é difícil estabelecer uma convenção universal para a alocação e a ordem de utilização de variáveis criadas especialmente para a passagem dos parâmetros. Administrar os nomes dessas variáveis também cria dificuldades. Finalmente, as variáveis globais ficam ocupando espaço o tempo todo, mas só são utilizadas dentro das sub-rotinas.

A passagem pela pilha, embora também seja mais lenta por exigir acesso à memória externa, é o método preferido devido à uniformidade de tratamento que se consegue e também por permitir naturalmente a chamada de uma sub-rotina dentro de outra, enquanto houver espaço na pilha.

7.3 Criação de variáveis locais

A segunda forma de utilização da pilha a ser discutida neste capítulo é a criação das assim chamadas *variáveis locais* ou *variáveis automáticas*. São variáveis criadas por uma sub-rotina em tempo de execução para armazenar resultados temporários e destruídas antes do retorno.

Como exemplo, suponha que a sub-rotina Show, utilizada nos exemplos anteriores para escrever mensagens na tela, seja alterada de modo a incluir uma contagem do número de caracteres escritos, retornando este número para o programa chamador no registrador AX. Suponha ainda que se deseja fazer a contagem do número de caracteres sem utilizar um dos registradores para armazená-la, mas sim uma posição de memória. Uma solução possível seria criar uma variável global apenas para esta finalidade. Entretanto, esta não é uma solução razoável, porque a variável só seria útil enquanto se executasse a sub-rotina, mas ocuparia espaço o tempo todo. O uso de uma variável local é mais indicado neste caso.

A criação de variáveis locais consiste em abrir uma brecha na pilha, correspondente ao espaço ocupado pelas variáveis que se pretende utilizar. Para tanto, subtrai-se do valor de SP o número de bytes necessários, logo após a preparação do registrador BP para apontar para os parâmetros passados. Por exemplo, para criar espaço para uma única variável do tipo *word*, o código da sub-rotina é:

```
PUSH BP
MOV BP,SP
SUB SP,2                ; Abre espaço para um word
```

O aspecto da pilha após a execução desta sequência por uma sub-rotina *near* que recebe dois parâmetros é o apresentado na figura 7.6.

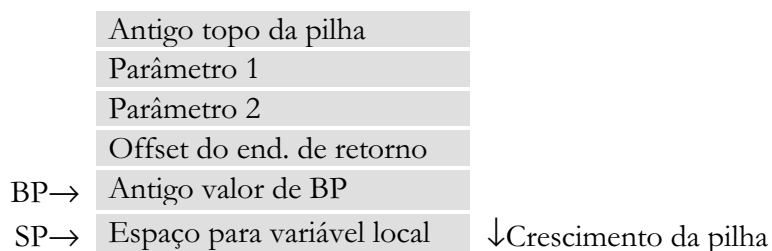


Fig. 7.6 – Criação de uma variável local

Após isso, a pilha pode ser utilizada normalmente, ficando o espaço criado disponível para armazenar valores temporários. É importante notar que:

- a variável local seria endereçada por BP-2. Se fossem criadas mais variáveis, seus endereços seriam BP-4, BP-6, e assim por diante;
- estes endereços são os mesmos para sub-rotinas *near* e *far*;
- as variáveis locais têm sempre deslocamentos negativos em relação a BP, enquanto os parâmetros têm sempre deslocamentos positivos;
- no final da sub-rotina, a brecha aberta na pilha precisa ser fechada, antes que se possa recuperar o valor de BP salvo na entrada. Para tanto, soma-se novamente a SP o valor subtraído na criação das variáveis. O final da sub-rotina ficaria assim:

```
ADD SP,2
POP BP
RET
```

A listagem a seguir mostra um programa com uma sub-rotina chamada SHOW, que utiliza uma variável local. Recebe dois parâmetros (endereços de duas strings a escrever na tela), conta o número de caracteres escritos e retorna este valor em AX. O valor retornado é então convertido para o sistema decimal e apresentado na tela.

```
;;;;;;;;;;;;;
;VarLoc.asm - Variaveis locais
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;
PILHA    SEGMENT STACK
         DW 40H DUP(?)
PILHA    ENDS

DADOS    SEGMENT
MSG1     DB 'ESTE PROGRAMA JUNTA DUAS MENASGENS $'
MSG2     DB 'EM UMA UNICA.$'
MSG3     DB 0DH,0AH,'A LINHA ACIMA TEM $'
MSG4     DB ' CARACTERES.$'
N        DB 00H,00H,'$'
DADOS    ENDS

CODE     SEGMENT
         ASSUME CS:CODE, DS:DADOS, SS:PILHA
INICIO:  MOV AX,DADOS
         MOV DS,AX
         MOV AX,OFFSET MSG1    ; ENDERECO DA MSG1
         PUSH AX
         MOV AX,OFFSET MSG2    ; ENDERECO DA MSG2
         PUSH AX
         CALL SHOW
         ADD SP,4              ; ELIMINA PARAMETROS DA PILHA
         MOV BL,10             ;
         DIV BL                 ; CONVERTE AX PARA DECIMAL
         OR AH,30H             ; ASCII
```

```
MOV N+1, AH          ; UNIDADE
OR AL, 30H           ; ASCII
MOV N, AL            ; DEZENA
MOV AH, 09H          ;
LEA DX, MSG3         ;
INT 21H              ; ESCRIVE MSG3
MOV AH, 09H          ;
LEA DX, N             ;
INT 21H              ; ESCRIVE VALOR DE N
MOV AH, 09H          ;
LEA DX, MSG4         ;
INT 21H              ; ESCRIVE MSG4
MOV AH, 4CH          ; TERMINA PROGRAMA
INT 21H

SHOW PROC NEAR
PUSH BP
MOV BP, SP
SUB SP, 2            ; VAR DE CONTAGEM TEMPORARIA
MOV WORD PTR [BP-2], 00H
MOV BX, [BP+6]       ; PRIMEIRA STRING
LOOP1: CMP BYTE PTR [BX], '$'
JZ OK1
INC WORD PTR [BP-2] ; CONTANDO CARACTERES DA MSG1
INC BX
JMP LOOP1
OK1: MOV AH, 09H
MOV DX, [BP+6]
INT 21H              ; ESCRIVE MSG1
MOV BX, [BP+4]       ; SEGUNDA STRING
LOOP2: CMP BYTE PTR [BX], '$'
JZ OK2
INC WORD PTR [BP-2] ; CONTANDO CARACTERES DA MSG2
INC BX
JMP LOOP2
OK2: MOV AH, 09H
MOV DX, [BP+4]
INT 21H              ; ESCRIVE MSG2
MOV AX, [BP-2]       ; VALOR A RETORNAR
ADD SP, 2            ; DESTROI VARIAVEL LOCAL
POP BP
RET
SHOW ENDP
CODE ENDS
END INICIO
```

7.3.1 Variáveis locais com inicialização

Uma forma alternativa de criar variáveis locais com inicialização utiliza a instrução PUSH, que coloca o valor correto na pilha. Como esta instrução não trabalha com endereçamento imediato no 8086, são necessários dois passos para criar a variável:

```
MOV AX,0000H      ; CRIA VARIÁVEL LOCAL  
PUSH AX           ; INICIALIZADA COM 0000H
```


8 Interrupções do 8086

8.1 Interrupções

As razões para a existência de um mecanismo que permita interromper o processamento normal de um programa expostas no capítulo 5 aplicam-se também no caso do 8086. Por isso, o texto assume que o leitor esteja a par da importância e da utilidade desse mecanismo e passa diretamente à explicação dos detalhes de sua implementação para este processador.

8.1.1 Características gerais

O 8086 pode tratar até 256 interrupções diferentes, numeradas de 00 a FFH. Como este número é significativamente maior do que as cinco interrupções possíveis no 8085, é compreensível que haja diferenças na implementação do mecanismo de interrupção do 8086. Estas diferenças aparecem nos seguintes pontos:

- nos endereços onde se localizam os tratadores de interrupção;
- no número de pinos dedicados à sinalização de interrupções de hardware;
- no mecanismo de habilitação / desabilitação das interrupções;
- nos mecanismos de chamada e retorno.

8.1.2 A tabela de vetores de interrupção

Ao contrário do que acontece no 8085, a localização dos tratadores de interrupção do 8086 não é pré-fixada no projeto do processador. Em vez disso, o 8086 faz uso da assim chamada *tabela de vetores de interrupção*. Esta contém 256 ponteiros do tipo far (os vetores), cada um correspondendo ao endereço inicial de um tratador. Com isso, os tratadores podem residir em qualquer parte da memória endereçável. Para encontrar o tratador correto

quando ocorre a interrupção de número i , o processador lê o i -ésimo vetor da tabela e coloca este valor no par CS:IP, desviando assim o processamento do programa para o tratador. É importante notar que a tabela não contém o tratador, mas apenas diz onde ele começa.

Para que isto funcione, o processador precisa saber onde fica a tabela. A solução adotada para este problema é simples: a tabela de vetores de interrupção inicia sempre no endereço fixo 0000H:0000H. Então os primeiros 4 bytes (0000H:0000H a 0000H:0003H) contêm o endereço inicial do tratador da interrupção 0, os 4 bytes seguintes contêm o endereço inicial do tratador da interrupção 1, e assim por diante. A figura 8.1 mostra, a título de exemplo, como ficaria o conteúdo dos primeiros 8 bytes da memória se os tratadores das interrupções 0 e 1 iniciassem, respectivamente, nos endereços 1234H:5678H e 0106H:1962H.

Endereço	Conteúdo
0000H:0000H	78H
0000H:0001H	56H
0000H:0002H	34H
0000H:0003H	12H
0000H:0004H	62H
0000H:0005H	19H
0000H:0006H	06H
0000H:0007H	01H

Fig. 8.1 – A tabela de vetores de interrupção do 8086

A disposição dos valores apresentados se explica através das seguintes regras, válidas para todos os processadores da linha 80x86:

- quando a memória contém um dado que ocupa mais de um byte, o byte menos significativo é armazenado no endereço menor, e o mais significativo, no endereço maior;
- o word correspondente ao segmento é considerado mais significativo do que o correspondente ao offset.

Desta forma, os offsets 0000H e 0001H da tabela contêm o valor do offset do tratador da interrupção 0, que é 5678H. O byte menos significativo, 78H, é armazenado primeiro, seguido do byte mais significativo, 56H. Em seguida vêm os dois bytes que compõem o valor do segmento. O endereço de início do tratador da interrupção 1 é armazenado de forma análoga.

A fim de facilitar a leitura dos ponteiros, é comum apresentar a tabela como na figura 8.2, em que os bytes da memória estão agrupados de quatro em quatro.

Embora esta seja a representação mais usada, é importante saber como se dá, de fato, o armazenamento. A figura mostra também que, no caso geral, o endereço tratador da i -ésima interrupção se localiza no offset $4i$ da tabela e que o endereço do último tratador reside nos offsets 03FCH a 03FFH. O comprimento total da tabela é de $256 \times 4 = 1024$ bytes = 1kB.

Endereço	Conteúdo
0000H:0000H	1234H:5678H
0000H:0004H	0106H:1962H
...	...
0000H: $4i$	Endereço inicial do tratador da interrupção i
...	...
0000H:03FCH	Endereço inicial do tratador da interrupção FFH

Fig. 8.2 – A tabela de vetores de interrupção do 8086

8.1.3 A sinalização das interrupções

A sinalização de eventos externos responsáveis pela interrupção do programa é feita através de dois pinos do processador, denominados NMI (*non-maskable interrupt*) e INT. O primeiro corresponde ao pino TRAP do 8085 e é de uso exclusivo para a chamada da interrupção 02, que não pode ser desabilitada e é utilizada em geral para sinalizar a ocorrência de um evento crítico, como por exemplo a queda iminente da alimentação. Todas as demais interrupções são sinalizadas através do pino INT, ao qual se liga normalmente um controlador de interrupções 8259, de forma semelhante à descrita no capítulo 5. É o 8259 que recebe a sinalização das interrupções propriamente ditas, e passa ao 8086 o número da solicitação recebida. De posse desta informação, o processador busca o endereço do tratador correspondente na tabela de vetores de interrupção.

8.1.4 Habilitando e desabilitando

A habilitação das interrupções é feita através do flag IF (*interrupt flag*), que reside na palavra de flags do processador. Este flag pode ser setado e resetado pelas instruções STI e CLI, respectivamente, e habilita / desabilita de uma só vez todas as interrupções, com exceção da interrupção 02 (NMI), que permanece sempre habilitada. O controle individual das interrupções não é

feito através 8086, mas sim mediante programação adequada do controlador de interrupção externo (8259).

8.1.5 O desvio para o tratador

O pedido de uma interrupção habilitada é atendido assim que termina a execução da instrução em curso no momento da solicitação. A fim de que o processamento do programa possa ser retomado após a execução do tratador, é necessário que o processador armazene automaticamente o endereço de retorno, que é o endereço da instrução seguinte à que estava sendo executada no momento da solicitação¹. Como o desvio a ser realizado altera o valor de CS:IP, é necessário armazenar os valores do segmento (CS atual) e do offset (IP, já corrigido para apontar para a instrução seguinte) desse endereço. Além disso, o 8086 salva também o valor atual dos flags. O armazenamento se dá na pilha, e os valores são armazenados nesta ordem: flags, CS, IP. Em seguida, o processador reseta o flag IF e executa o desvio. Desta forma, a execução do tratador inicia com as interrupções desabilitadas, o que evita que o mesmo tratador seja chamado várias vezes, de forma recursiva, por um único pedido de interrupção.

O retorno é feito mediante a instrução IRET, que finaliza o tratador, recuperando os valores salvos na pilha por ocasião do desvio.

O armazenamento dos flags tem uma consequência digna de nota: não é necessário habilitar as interrupções dentro do tratador, a menos que se deseje atender a outros pedidos de interrupção durante sua execução. Não existe o risco de que as interrupções fiquem desabilitadas para sempre, porque os flags salvos na pilha incluem o valor que o flag IF tinha antes do desvio. Como os flags são recuperados pela instrução IRET, o estado anterior se restabelece automaticamente após o retorno.

8.1.6 Interrupções de software

A instrução INT *i*, onde *i* é um número de 00 a FFH, faz com que o processador proceda exatamente como se tivesse recebido uma solicitação da interrupção *i*. Por isso, esta instrução recebe o nome de *interrupção de software*. A menos do armazenamento dos flags na pilha, executar a instrução INT *i* tem o mesmo efeito de chamar o tratador dessa interrupção como uma sub-rotina,

¹ Uma exceção a esta regra são as instruções de manipulação de strings, discutidas mais adiante.

mas com uma importante diferença: o endereço da sub-rotina não é dado de forma explícita, e isto cria um mecanismo de chamada indireta de sub-rotinas, muito útil em situações em que a sub-rotina a ser chamada pode mudar de endereço sem que se queira recompilar o programa chamador por causa disso. Um exemplo de aplicação desse mecanismo são os serviços do DOS (v. anexo 4). A utilização das interrupções de software permite que as sub-rotinas de serviço (os tratadores das interrupções chamadas) possam mudar de lugar de uma versão para outra do sistema operacional, sem que isso implique na recompilação dos aplicativos do usuário. Para manter o serviço funcionando, o DOS só precisa atualizar a tabela de vetores de interrupção.

8.1.7 Interrupções reservadas

Algumas interrupções têm finalidades especiais para o 8086 e por isso são consideradas reservadas. A primeira delas é a interrupção 0, que é gerada automaticamente quando o divisor passado para uma instrução de divisão for igual a zero. Desta forma, o tratador da interrupção 0 pode ser visto como um tratador de exceção para este erro, chamado automaticamente.

O tratador da interrupção 1 é chamado automaticamente sempre que o flag TF (*trap flag*) estiver setado. Este mecanismo foi criado para facilitar a implementação de depuradores, que permitem que se executem as instruções de um programa passo a passo, parando após cada uma delas. O princípio de utilização consiste em setar esse flag imediatamente antes da instrução que se pretende executar desta forma. Assim que a instrução é executada, o tratador da interrupção 1 é chamado e pode, então, apresentar informações sobre o estado atual do processador. Para que o tratador não seja ele próprio trancado pela execução, o processador reseta o flag TF antes de desviar para o tratador.

A interrupção 3 é uma interrupção cuja codificação ocupa apenas 1 byte em vez dos dois bytes ocupados normalmente pela instrução INT *i*. Isto serve novamente aos depuradores, que a utilizam na criação de *breakpoints*², substituindo o byte que está num determinado endereço pelo opcode da interrupção 3. Uma vez alcançado este endereço, o programa é desviado para o tratador correspondente, que executa as ações necessárias.

Finalmente, o tratador da interrupção 4 é chamado pela instrução INTO (*interrupt if overflow*) sempre que esta for executada com o flag OF (*overflow flag*)

² Um breakpoint é um recurso de depuração que permite executar o programa sob teste até um endereço especificado e então pará-lo, a fim de examinar em detalhe o que está acontecendo nesse ponto da execução.

setado. Isto permite ao programa reagir de forma conveniente a erros de overflow de divisão (os casos em que o resultado de uma divisão é finito, mas não cabe no registrador de destino).

Além das interrupções descritas, a Intel considera reservadas as interrupções 05H a 1FH. Embora estas não sejam especiais no 8086, não se deve utilizá-las, pois elas têm funcionalidade específica nos demais processadores da família 80x86. Utilizá-las para fins particulares pode implicar em perda de compatibilidade do software com estes processadores.

A tabela 8.1 resume as interrupções especiais do 8086.

Número	Offset do vetor	Uso
00	0000H	Detecção de divisão por zero
01	0004H	Execução passo a passo
02	0008H	Interrupção não mascarável
03	000CH	Breakpoints
04	0010H	Overflow
05 – 1FH	0014H – 007CH	Reservado

Tab. 8.1 – Interrupções reservadas do 8086

9 Manipulação de strings

9.1 Introdução

O 8086 conta com um grupo de instruções que dão suporte à realização de tarefas que aparecem com frequência quando se manipulam cadeias de caracteres (*strings*). Uma vez que a representação de uma cadeia de caracteres na memória é a mesma de uma tabela, estas instruções são igualmente úteis na manipulação de tabelas genéricas.

A principal vantagem do emprego das instruções de manipulação de strings está no ganho de velocidade que proporcionam. Existe também uma pequena redução no tamanho do programa, mas esta vantagem é geralmente insignificante quando comparada ao aumento de desempenho.

O objetivo deste capítulo é o de apresentar essas instruções, exemplificar seu uso e mostrar de onde vem esse ganho de desempenho.

9.2 As instruções

As instruções disponíveis dão suporte às seguintes operações:

- cópia de uma região da memória para outra;
- comparação de duas strings / tabelas (localização do primeiro elemento igual ou diferente);
- varredura (busca do primeiro caracter igual ou diferente de um caracter de comparação);
- leitura dos elementos de uma string, um por vez;
- inicialização de tabelas.

A tabela 9.1 apresenta as instruções de manipulação de strings existentes no 8086, sua aplicação e a classificação dos argumentos necessários.

Mnemônico	Aplicação	Argumentos
MOVS / MOVSB / MOVSW	Cópia	Destino, origem
CMPS / CMPSB / CMPSW	Comparação	Destino, origem
SCAS / SCASB / SCASW	Varredura	Destino
LODS / LODSB / LODSW	Leitura	Origem
STOS / STOSB / STOSW	Inicialização	Destino

Tab. 9.1 – As instruções de manipulação de strings

Note que as instruções de cópia e de comparação atuam sobre duas strings diferentes, enquanto as demais atuam sobre uma única string. Todas têm em comum as seguintes características:

- o argumento de origem está localizado sempre em DS:[SI];
- o argumento de destino está localizado, por default, em ES:[DI]; o segmento pode ser mudado para CS, DS ou SS com um prefixo de modificação de segmento; o offset é sempre dado por DI;
- podem agir tanto sobre bytes quanto sobre words;
- dependendo do valor do flag de direção (DF), as strings são manipuladas do início para o fim (DF = 0) ou do fim para o começo (DF = 1);
- quando uma instrução de manipulação de strings é executada, os registradores SI e DI, se utilizados, são ajustados para apontarem para o próximo elemento da string em questão. Este ajuste pode ser um incremento (DF = 0) ou decremento (DF = 1) de uma unidade (quando a instrução age sobre bytes) ou de duas unidades (idem, words);
- todas as instruções ocupam apenas um byte, precedido de um prefixo de modificação de segmento de destino, quando for o caso.

A tabela 9.1 mostra que existem três mnemônicos diferentes para cada uma das aplicações possíveis. O primeiro mnemônico termina sempre pela letra ‘S’, que denota *string*. Esta forma dos mnemônicos exige argumentos que permitam ao assembler concluir se a instrução se refere à manipulação de bytes ou de words. Por exemplo, a instrução MOVS pode ser utilizada das seguintes formas:

```
MOVS BYTE PTR ES:[DI], [SI]
```

ou

```
MOVS WORD PTR ES:[DI], [SI]
```

significando que a operação a ser realizada é a cópia do byte (no primeiro caso) ou do word (no segundo caso) em DS:[SI] para ES:[DI]. É importante notar que, embora a instrução seja escrita com dois argumentos, sua codificação continua sendo feita em um único byte; os argumentos servem apenas para o que assembler possa decidir se a instrução deve agir sobre bytes ou words e se

há necessidade de algum prefixo de modificação do segmento de destino. Nos casos acima, que utilizam o segmento default, pode-se escrever simplesmente

```
MOVSB
```

ou

```
MOVSW
```

respectivamente, com o mesmo resultado. A forma longa, com argumentos explícitos, só é obrigatória quando se quer mudar o segmento de destino, como em

```
MOVS BYTE PTR DS:[DI],[SI]
```

ou

```
MOVS WORD PTR DS:[DI],[SI]
```

Nestes exemplos, a cópia se faz dentro de um mesmo segmento.

9.3 Um exemplo — cópia de tabelas

Seja a tarefa de copiar o conteúdo de uma tabela de 1 kB, localizada no offset TAB1 do segmento DATA1, para uma tabela de igual tamanho, localizada no offset TAB2 de um segmento chamado DATA2. O programa abaixo mostra como esta cópia poderia ser feita, primeiramente sem a utilização das instruções de manipulação de strings.

```
TABSIZE EQU 400H
PILHA    SEGMENT STACK
         DB 128 DUP(?)
PILHA    ENDS
DATA1    SEGMENT
TAB1     DB TABSIZE DUP(55H)
DATA1    ENDS
DATA2    SEGMENT
TAB2     DB TABSIZE DUP(00H)
DATA2    ENDS
CODE     SEGMENT
         ASSUME CS:CODE,DS:DATA1,ES:DATA2
START:   MOV AX,DATA1
         MOV DS,AX
         MOV AX,DATA2
         MOV ES,AX
         LEA SI,TAB1
         LEA DI,TAB2
         MOV CX,TABSIZE
LOOP:    MOV AL,[SI]
         MOV ES:[DI],AL
```

```
INC SI
INC DI
DEC CX
JNZ LOOP
...
```

O código apresentado pode ser melhorado mediante utilização da instrução `MOVS` que, de acordo com a seção 9.2, substitui as quatro primeiras linhas do loop. Para ter certeza de que a cópia se dê na direção certa (incremento de SI e DI), é preciso colocar antes uma instrução `CLD`. O código fica assim:

```
START:  MOV AX, DATA1
        MOV DS, AX
        MOV AX, DATA2
        MOV ES, AX
        LEA SI, TAB1
        LEA DI, TAB2
        MOV CX, TABSIZE
        CLD
LOOP:   MOVS
        DEC CX
        JNZ LOOP
...
```

Neste ponto é possível começar a ver como a utilização das instruções de manipulação de strings afeta o tamanho do programa e seu desempenho. A redução no tamanho do programa se dá porque uma instrução de manipulação de strings substitui várias instruções convencionais.

O ganho de desempenho merece uma análise mais detalhada. Ele não vem de algum recurso especial da instrução de manipulação de strings em si, pois esta precisa ler o byte a ser copiado da sua posição de origem, escrevê-lo na sua posição de destino e incrementar os registradores SI e DI, da mesma forma que o código sem a instrução de manipulação de strings. Estas operações, principalmente os acessos à memória, exigem um tempo que não há como reduzir.

O ganho de desempenho vem do fato de que, em geral, as instruções que fazem a manipulação são executadas um grande número de vezes. No caso do exemplo acima, as linhas que compõem o loop são executadas tantas vezes quantos forem os bytes copiados (400H ou 1024). É preciso lembrar agora que, quando executa um loop, o processador tem de ler e decodificar essas instruções a cada vez que são executadas, e isso significa que, no primeiro caso, precisa executar $6 \times 1024 = 6144$ instruções, enquanto que no segundo são executadas $3 \times 1024 = 3072$. Cada instrução dessas exige a leitura de pelo menos 1 byte da memória, e portanto há mais tempo sendo gasto na leitura de instruções do que na cópia das tabelas. Com a redução desse tempo à metade no segundo caso, começa a aparecer a vantagem das instruções de manipulação

de strings. Este tempo pode ser reduzido ainda mais, através do emprego dos assim chamados prefixos de repetição, analisados a seguir.

9.4 Os prefixos de repetição

A fim de facilitar a repetição de uma instrução de manipulação de strings, o 8086 oferece os *prefixos de repetição*, que são códigos colocados imediatamente antes dessas instruções e que causam sua repetição até que uma dada condição de parada seja atingida.

Todos os prefixos de repetição testam CX antes de executar a instrução que segue. Se CX for zero, a instrução não é executada. Caso contrário, CX é decrementado de uma unidade e a instrução é executada. Se a instrução em questão for uma instrução de comparação (CMPS ou SCAS), os flags são afetados de acordo com o resultado da comparação feita.

A tabela 9.2 apresenta os prefixos de repetição disponíveis e as condições de parada correspondentes.

Prefixo	Condição de parada
REP	CX = 0
REPE / REPZ	CX = 0 ou ZF = 0
REPNE / REPNZ	CX = 0 ou ZF = 1

Tab. 9.2 – Prefixos de repetição do 8086

O prefixo mais simples é REP (*repeat*), que simplesmente repete a instrução em questão CX vezes. O prefixo REPE e seu sinônimo REPZ (*repeat while equal / zero*) repetem a instrução em questão enquanto o *zero flag* permanecer setado. Se a instrução for de comparação, a repetição continuará enquanto houver igualdade entre os elementos comparados. A repetição cessa quando o flag deixar de ser setado ou então quando CX chega a zero. Este prefixo pode ser usado com a instrução CMPS, para fazer com que o processador compare duas tabelas e pare assim que encontrar a primeira desigualdade entre elas. Considerando que o *direction flag* esteja em zero, os offsets dos elementos desiguais serão então SI-1 e DI-1 (se os elementos forem bytes) ou SI-2 e DI-2 (se forem words). A subtração de 1 ou 2 é necessária porque SI e DI são automaticamente ajustados para apontar para o elemento seguinte, independentemente do resultado da comparação. Se DF estiver setado, é necessário somar em vez de subtrair.

O prefixo REPNE ou seu sinônimo REPNZ (*repeat while not equal / not zero*) é o dual do anterior e repete a instrução em questão enquanto o *zero flag*

permanecer zerado. Este prefixo pode ser utilizado para encontrar uma igualdade entre duas tabelas.

Podemos agora melhorar ainda mais o código apresentado na exemplo da seção 9.3, trocando as linhas do loop do segundo caso por uma única linha. O segmento de código fica então assim:

```
START:  MOV AX,DATA1
        MOV DS,AX
        MOV AX,DATA2
        MOV ES,AX
        LEA SI,TAB1
        LEA DI,TAB2
        MOV CX,TABSIZE
        CLD
REP     MOVSB
        ...
```

Note que agora não existe mais o loop para fazer a repetição da instrução MOVSB. Isso significa que ela é lida apenas uma única vez (2 bytes ao todo, considerando o prefixo de repetição), e não mais 6144 ou 3072 vezes, como nos casos discutidos na seção 9.3. Agora o tempo de execução da instrução corresponde praticamente todo à realização de trabalho útil, pois o tempo gasto na leitura e decodificação da instrução é desprezível quando comparado ao tempo necessário para a cópia da tabela. É assim que as instruções de manipulação de strings conseguem proporcionar um ganho de velocidade considerável ao processador.

9.5 Ocorrência de interrupções

Em geral, quando acontece um pedido de interrupção, o processador termina de executar a instrução em andamento e só depois atende esse pedido. A execução das instruções de manipulação de strings, porém, pode ser muito mais demorada do que a das instruções comuns. Por isso, essas instruções são interrompíveis. Quando acontece uma interrupção durante o processamento de uma instrução de manipulação de strings, processador termina de fazer a operação que está em andamento (cópia, comparação, etc.) e interrompe o loop determinado pela contagem em CX, para atender o pedido de interrupção. Depois de atendido o pedido, o processador retorna ao ponto em que havia interrompido o processamento da instrução de manipulação de strings. Desta forma, a presença dessas instruções não atrapalha o atendimento de pedidos de interrupção.

9.6 Exemplos de aplicação

Esta seção apresenta alguns exemplos de aplicação das instruções de manipulação de strings, complementando o exemplo de cópia de tabelas discutido nas seções anteriores. Outros exemplos podem ser encontrados no capítulo 21.

9.6.1 Comparação de tabelas

A comparação de duas tabelas pode ser feita com auxílio da instrução CMPS ou de suas formas abreviadas CMPSB e CMPSW. Estas instruções comparam as tabelas subtraindo o byte ou word endereçado pelo registrador que varre a tabela de origem do byte ou word endereçado pelo registrador que varre a tabela de destino, respectivamente. Os registradores são ajustados (incrementados ou decrementados, dependendo do valor do *direction flag*) de acordo com o tamanho do elemento comparado. A subtração é utilizada apenas para setar os flags; o resultado é descartado.

O trecho de código a seguir compara duas tabelas, TAB1 e TAB2, localizadas num mesmo segmento, endereçado pelo registrador DS. A comparação termina quando se encontra uma igualdade entre bytes de mesmo offset (em relação ao início das tabelas) ou então quando se chega ao final das tabelas. O *zero flag* é testado após a instrução de comparação para saber se houve igualdade ou não. É interessante notar que o teste deste flag é válido mesmo quando CX chega a zero, porque CX é sempre decrementado antes da execução da instrução, de modo que o estado do flag corresponde sempre ao resultado da comparação. Quando uma igualdade é encontrada, o registrador SI é decrementado de uma unidade para que volte a conter o offset do byte que causou a parada da repetição.

```
START:  MOV AX,DADOS
        MOV DS,AX
        LEA SI,TAB1
        LEA DI,TAB2
        CLD
REPE    CMPSB
        JZ IGUAL
        JMP DIFERENTE
IGUAL:  DEC SI
        ...
```

9.6.2 Varredura de tabelas

A varredura de uma tabela em busca de um determinado byte ou word pode ser feita com o auxílio da instrução SCAS ou de suas formas abreviadas SCASB e SCASW. Estas instruções comparam o elemento endereçado pelo registrador que varre a tabela (DI) com o registrador AL ou AX, respectivamente, e ajustam DI de uma ou duas unidades, conforme o caso. Note que, de acordo com a tabela 9.1, a tabela é endereçada como um operando de destino, e que por isso o registrador de segmento default é ES, mas pode ser mudado com um prefixo de modificação de segmento. O registrador de offset é, obrigatoriamente, DI.

O trecho de código abaixo varre a tabela TAB1 em busca do word 1234H. Caso o encontre, o registrador DI é decrementado de duas unidades, para que aponte para o word encontrado.

```
START:  MOV AX,DADOS
        MOV ES,AX
        LEA DI,TAB1
        CLD
        MOV AX,1234H
REPNE   SCASW
        JZ IGUAL
        JMP DIFERENTE
IGUAL:  SUB DI,2
        ...
```

9.6.3 Leitura de tabelas

A leitura de uma tabela pode ser agilizada com a instrução LODS ou de suas formas abreviadas LODSB e LODSW. Estas instruções lêem um byte ou um word para o registrador AL ou AX, respectivamente, do endereço contido em SI e ajustam este registrador em uma ou duas unidades, conforme o caso. Dificilmente são utilizadas em conjunto com algum prefixo de repetição, pois em geral é necessário processar de alguma forma cada elemento lido antes de ler o próximo. Mesmo assim, existe o ganho de se conseguir, numa instrução só, ler o byte ou word e ainda incrementar o registrador, o que diminui o número de instruções num loop. Note que, de acordo com a tabela 9.1, o argumento é considerado de origem, e por isso o segmento e o offset são sempre DS e SI, respectivamente.

O trecho de código abaixo utiliza a instrução LODSB para fazer em BL a soma dos bytes de uma cadeia de caracteres STR1, sem considerar o “vai 1”. Esta

operação pode ser útil no cálculo do *checksum* de uma string após sua recepção por um meio de transmissão.

```
    ...  
    LEA SI,STR1  
    CLD  
    MOV BL,00H  
    MOV CX,STRSIZE ; COMPRIMENTO DA STRING  
LOOP: LODSB  
    ADD BL,AL  
LOOPNZ LOOP
```

Aqui aparece um outro prefixo de repetição do 8086, LOOPNZ. Este, assim como seu sinônimo LOOPNE, substitui as linhas

```
    DEC CX  
    JNZ LOOP
```

fazendo assim com que o processador tenha uma instrução a menos para ler no loop.

9.6.4 Inicialização de tabelas

A instrução STOS, assim como suas formas abreviadas STOSB e STOSW, pode ser utilizada para inicializar uma tabela com um dado valor. Estas instruções escrevem o byte ou word contido em AL ou AX, respectivamente, no endereço contido no registrador que varre a tabela e ajustam este registrador em uma ou duas unidades, conforme o caso.

Note que, de acordo com a tabela 9.1, a tabela é endereçada como um operando de destino, e que por isso o registrador de segmento default é ES, mas pode ser mudado com um prefixo de modificação de segmento. O registrador de offset é, obrigatoriamente, DI.

O trecho de código a seguir utiliza a instrução STOSW para inicializar uma tabela de 1 kB (400H bytes ou 200H words) com o valor 3535H.

```
    ...  
    LEA DI,STR1  
    CLD  
    MOV CX,200H  
    MOV AX,3535H  
REP    STOSW
```




Parte II – Laboratório





10 O Abacus

10.1 O simulador e o assembler

O Abacus é um simulador e assembler para o microprocessador 8085, desenvolvido especialmente para acompanhar este livro. Sua interface é mostrada na figura 10.1.

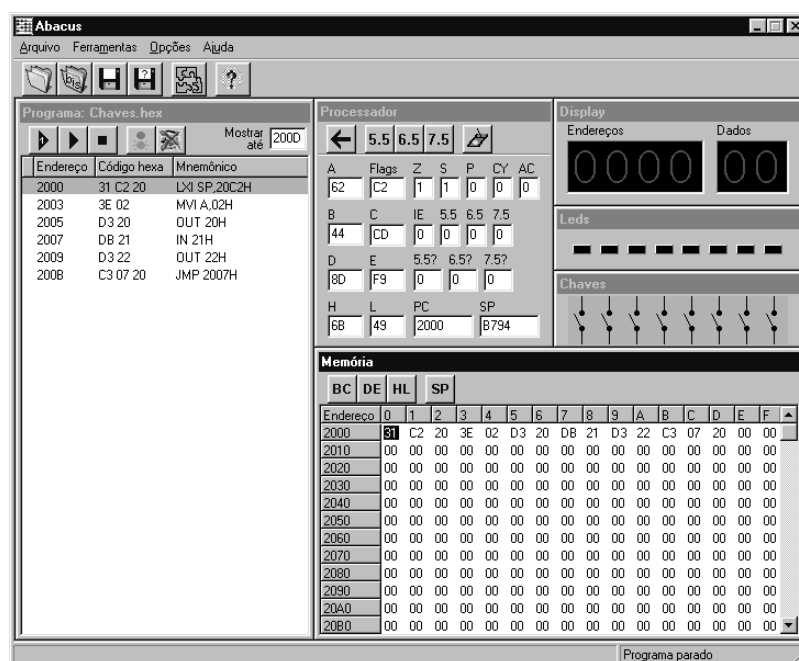


Fig. 10.1 – O Abacus

Com o Abacus, é possível executar praticamente qualquer programa que caiba dentro da região de RAM simulada e acompanhar as mudanças no processador e na memória durante a execução. É possível visualizar o comportamento da pilha e a varredura de tabelas, criar breakpoints para interromper a execução

em qualquer ponto, alterar o estado do processador e da memória, executar instruções passo a passo e gerar interrupções.

Como dispositivos de entrada podem ser utilizados o próprio teclado do computador ou um conjunto de chaves, simuladas pelo software; a saída pode ser feita em um display numérico ou em um conjunto de leds.

O Abacus pode também ser utilizado no modo assembler, em que é possível criar programas para o 8085 a partir de uma tabela de mnemônicos. A seção 10.3.7 traz mais detalhes sobre este modo.

Com estes elementos, é possível resolver todos os exercícios propostos para o 8085 e ainda desenvolver muitos outros, que ilustram não só o funcionamento deste microprocessador, mas principalmente conceitos gerais da disciplina de Microprocessadores.

10.2 A memória

O Abacus simula para o processador 8085 uma região de 1 kB de memória RAM, que se estende do endereço 2000H ao endereço 23FFH. Qualquer programa a ser executado deve ser colocado nesta região de memória. Tentativas de acesso a endereços inválidos páram a execução do programa e geram uma mensagem de aviso.

A região que vai de 0000H a 1FFFFH é considerada memória ROM e não pode ser visualizada no simulador. Apesar disso, o Abacus se comporta como se houvesse, em alguns endereços, sub-rotinas que o usuário pode chamar e, em outros, instruções de desvio para a região de RAM (v. seção 10.4.3).

10.3 As janelas

A interface do Abacus consiste das seguintes janelas principais, que são descritas em detalhe nesta seção:

- Programa;
- Processador;
- Memória;
- Display;
- Leds;
- Chaves;
- Assembler.

10.3.1 A janela Programa

Esta janela, mostrada na figura 10.2, permite visualizar o programa carregado na memória RAM simulada do Abacus. A janela é dividida em três colunas. A primeira coluna mostra os endereços em que iniciam as instruções presentes na memória, a segunda coluna os códigos hexadecimais dessas instruções e a terceira os mnemônicos correspondentes.



Endereço	Código hexa	Mnemônico
2000	31 C2 20	LXI SP,20C2H
2003	3E 02	MVI A,02H
2005	D3 20	OUT 20H
2007	DB 21	IN 21H
2009	D3 22	OUT 22H
200B	C3 07 20	JMP 2007H

Fig. 10.2 – Detalhe da janela Programa

O cursor

A janela Programa tem também um cursor, uma linha azul-clara que destaca o endereço contido no registrador PC e que corresponde à próxima instrução a ser executada.

Às vezes é necessário modificar o ponto de continuação da execução de um programa durante um teste. Isto corresponde a uma mudança da posição do cursor e pode ser conseguido com um duplo clique sobre o endereço desejado na janela Programa.

Breakpoints

Um clique com o botão esquerdo do mouse à esquerda de uma linha de programa cria um breakpoint nessa linha. A cor da linha muda para vermelho, para indicar a presença do breakpoint. A passagem por um breakpoint interrompe a execução do programa. Isto permite executar um programa até que se atinja um determinado ponto e então parar, para examinar em detalhe o que acontece. Por isso, os breakpoints são de grande utilidade no teste e na depuração. Pode haver até 10 breakpoints ativos simultaneamente. Para remover um breakpoint, basta clicar novamente à esquerda da linha. Para remover todos os breakpoints simultaneamente, clique no botão



“Remover breakpoints”

Execução de um programa

O controle da execução de um programa é feito através dos seguintes botões:



“Executar”: executa o programa a partir da posição do cursor,



“Parar”: interrompe a execução e



“Executar passo a passo”: executa uma instrução de cada vez.

Atualização da janela

A execução de um programa pode modificar o conteúdo da memória. Quando acontece uma modificação em um endereço que está sendo mostrado na janela de programa, a decodificação das instruções afetadas precisa ser atualizada. Esta atualização é feita automaticamente cada vez que a execução pára, e normalmente não é feita durante a execução. No entanto, é possível fazer com que esta atualização aconteça também durante a execução, clicando em



“Atualizar durante execução”.

O campo “Mostrar até”

Em princípio, a janela Programa poderia mostrar sempre toda a memória RAM simulada. Isto criaria, porém, um certo desconforto na utilização da sua barra de rolagem, pois bastaria um pequeno deslocamento do cursor para causar um salto de um grande número de linhas de programa. Ao mesmo tempo, geralmente se trabalha com programas curtos, de maneira que quase nunca é importante visualizar a memória toda através desta janela.

A fim de evitar esse desconforto, a janela Programa mostra, por default, apenas os endereços correspondentes ao arquivo carregado. O último endereço que pode ser visualizado nesta janela aparece no campo “Mostrar até”, que pode ser alterado pelo usuário para modificar o tamanho da região mostrada. Se o endereço colocado em “Mostrar até” não corresponder ao último byte de uma instrução, então a visualização pára na última instrução que pode ser mostrada por inteiro.

Enquanto não há programa carregado, são mostrados os primeiros 64 bytes da memória.

A janela Programa em modo assembler

A execução de programas não é permitida no modo assembler. Por isso, os botões descritos acima não são visíveis. No lugar deles, aparecem dois outros, úteis para ajudar na edição de programas:



“Inserir um byte”: insere um byte igual a 00 na posição do cursor;



“Remover um byte”: remove o primeiro byte da linha do cursor.

10.3.2 A janela Processador

Esta janela, mostrada na figura 10.3, permite visualizar o estado do processador 8085. Estão presentes os registradores e flags, além dos bits da máscara de interrupções. Os valores de todos os registradores são hexadecimais. O valor decimal correspondente aos registradores de A até L pode ser visualizado rapidamente, bastando para isso manter o cursor do mouse em repouso por alguns instantes sobre o valor hexadecimal.



Fig. 10.3 – A janela Processador

A janela conta ainda com os seguintes botões:



“Reset”: causa um reset do processador;



“RST 5.5”: solicitação da interrupção 5.5;



“RST 6.5”: solicitação da interrupção 6.5;



“RST 7.5”: solicitação da interrupção 7.5;



“TRAP”: solicitação da interrupção “trap”.

Para esclarecer dúvidas sobre os elementos desta janela, consulte os capítulos da parte teórica que tratam do 8085. Os registradores e flags que aparecem na janela correspondem ao modelo de programação. O capítulo 15 traz ainda alguns detalhes adicionais sobre o funcionamento dos botões relacionados às interrupções.

10.3.3 A janela Memória

Esta janela, mostrada na figura 10.4, permite visualizar e modificar o conteúdo de qualquer byte da memória RAM simulada pelo Abacus.

Memória																
	BC	DE	HL	SP												
Endereço	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2000	81	C2	20	3E	02	D3	20	DB	21	D3	22	C3	07	20	00	00
2010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Fig. 10.4 – A janela Memória

Seguidores de ponteiros

Os botões da barra de ferramentas desta janela ativam um mecanismo que permite acompanhar programas que trabalham com endereçamento indireto, por exemplo com varredura de tabelas. Nestes casos, um par de registradores contém um endereço de memória, e diz-se então que este par de registradores aponta para o byte que está nesse endereço. Quando um dos botões

- BC** “Seguir BC”,
- DE** “Seguir DE” ou
- HL** “Seguir HL”

é ativado, o byte endereçado é destacado. Desta forma, um programa que varie um desses endereços para acessar tabelas dados de forma indireta produz um efeito de animação que permite compreender melhor seu funcionamento.

De forma análoga, o botão

SP “Seguir SP”

destaca o byte que está no topo da pilha. Os bytes são destacados em cores diferentes, para evitar confusão quando mais de um botão está ativo.

10.3.4 A janela Display

O Abacus inclui um display, que pode ser utilizado como dispositivo de saída e é mostrado na figura 10.5. Sua utilização é feita através de sub-rotinas simuladas em ROM, explicadas no capítulo 12.

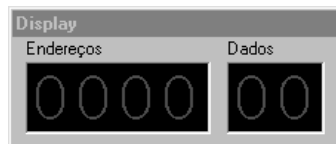


Fig. 10.5 – A janela Display

10.3.5 A janela Leds

Esta janela simula um conjunto de oito leds, dispostos horizontalmente conforme a figura 10.6. O led mais à esquerda corresponde ao bit mais significativo de uma porta de saída que, quando recebe um byte de um programa, faz acender os leds nas posições cujos bits forem iguais a 1.

O capítulo 16 dá mais detalhes sobre como trabalhar com os leds.



Fig. 10.6 – A janela Leds

10.3.6 A janela Chaves

Esta janela simula um conjunto de chaves, mostradas na figura 10.7. Estas chaves podem ser lidas a partir de uma porta de entrada. O processador lê um byte dessa porta, cujos bits estarão em 0 ou 1 conforme as chaves estejam abertas ou fechadas, respectivamente. A chave mais à esquerda corresponde ao bit mais significativo.

Detalhes sobre como trabalhar com as chaves são dados no capítulo 16.

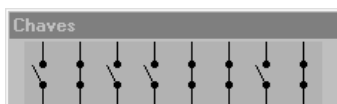


Fig. 10.7 – A janela Chaves

10.3.7 A janela Assembler

O modo assembler é ativado através do botão



“Exibir assembler”

da barra de ferramentas da janela principal do Abacus. Neste modo, as janelas Processador, Leds, Display e Chaves dão lugar à janela Abacus Assembler, mostrada na figura 10.8.

Abacus Assembler										
ACI	ADC	ADD	ADI	ANA	A	ALL	CC	CM	CMA	CMC
CMP	CNC	CNZ	CP	CPE	B	PO	CZ	DAA	DAD	DCR
DCX	DI	EI	HLT	IN	C					
JNZ	JP	JPE	JPD	JZ	D	XX	JC	JM	JMP	JNC
					E					
					H	AX	LHLD	LXI	MOVA	MOVB
MOV C	MOV D	MOV E	MOV H	MOV L	L	VI	NOP	ORA	ORI	OUT
PCHL	POP	PUSH	RAL	RAR	M					
						ET	RIM	RLC	RM	RNC
RNZ	RP	RPE	RPO	RRC		RZ	SBB	SBI	SHLD	SIM
SPHL	STA	STAX	STC	SUB	SUI	XCHG	XRA	XRI	XTHL	

Fig. 10.8 – O Assembler do Abacus

A edição de programas pode então ser feita selecionando-se com o mouse as instruções desejadas. Quando se seleciona uma instrução, o opcode correspondente é colocado na memória e as janelas Programa e Memória são atualizadas de acordo.

Instruções em que é preciso escolher um registrador, como por exemplo ANA <reg>, abrem automaticamente um pequeno menu, como o que aparece na figura anterior. Instruções que precisam de argumentos numéricos, como por exemplo JMP <endereço>, colocam na memória apenas o opcode; os bytes que formam o endereço podem então ser introduzidos através do teclado.

Programas editados desta forma podem ser salvos em disco, conforme descrito na seção 10.4.1.

10.4 Os menus

A interface do Abacus conta ainda com os seguintes menus, descritos em detalhe a seguir:

- Arquivo;
- Ferramentas;
- Opções;
- Ajuda.

10.4.1 O menu Arquivo

O menu Arquivo do Abacus abriga os seguintes itens:

Abrir

Permite abrir arquivos existentes. Os arquivos utilizados devem ter a extensão .hex e o formato descrito na seção 11.2.3. O nome do programa aberto aparece na barra de título da janela Programa. O mesmo efeito pode ser conseguido clicando-se o botão



”Abrir arquivo”,

da barra de ferramentas principal do Abacus.

Recarregar

Carrega novamente o último arquivo aberto. Esta opção é bastante útil durante o teste e a depuração de programas, em que muitas vezes se chega a uma situação em que se deseja recomençar o teste do início, dispensando qualquer modificação feita. A maneira mais segura de fazer isto é recarregar o programa. O mesmo efeito pode ser conseguido clicando-se o botão



“Recarregar arquivo”,

da barra de ferramentas principal do Abacus.

Salvar

Salva o programa atual com o mesmo nome com o qual foi carregado. Por programa atual entende-se o conteúdo da faixa de memória visível na janela Programa (do início da RAM até o endereço contido em “Mostrar até”). O mesmo efeito pode ser conseguido clicando-se o botão



“Salvar”,

da barra de ferramentas principal do Abacus.

Salvar como

Semelhante ao item anterior. Salva o programa atual com um nome que pode ser escolhido pelo usuário. O mesmo efeito pode ser conseguido clicando-se o botão



“Salvar como”,

da barra de ferramentas principal do Abacus.

Sair

Encerra o Abacus.

10.4.2 O menu Ferramentas

Este menu tem apenas o item

Assembler,

que tem a mesma funcionalidade do botão



“Exibir assembler”,

descrito na seção 10.3.7.

10.4.3 O menu Opções

Este menu abriga os seguintes itens:

Sub-rotinas em ROM

Este item leva à caixa de diálogo mostrada na figura 10.9.

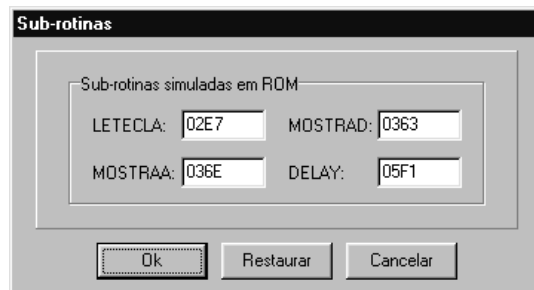


Fig. 10.9 – A caixa de diálogo Sub-rotinas

O Abacus inclui algumas sub-rotinas simuladas em ROM, cujos endereços iniciais podem ser modificados pelo usuário. A facilidade de modificar estes endereços existe apenas para os casos em que se deseje utilizar o Abacus para simular a execução de programas em um hardware específico, com sub-rotinas análogas, mas localizadas em endereços diferentes.

As listagens contidas neste livro utilizam sempre os endereços default, e por isso não é necessário modificar estas configurações. Caso isso seja feito, os valores default podem ser recuperados através do botão “Restaurar”.

Desvios de tratadores para RAM

Este item leva à caixa de diálogo mostrada na figura 10.10.



Fig. 10.10 – A caixa de diálogo Tratadores

Os endereços de desvio das interrupções do processador 8085 não correspondem à região de memória RAM simulada pelo Abacus. Isto é pouco prático, pois para utilizar as interrupções é preciso poder modificar o conteúdo da memória nesses endereços. Para contornar este problema, o Abacus permite reprogramar os desvios. O programa se comporta então como se houvesse, nos endereços de desvio do processador, instruções JMP para os novos endereços em RAM.

Os valores default são utilizados ao longo do livro e não precisam ser alterados para a realização dos exercícios. Caso sejam, podem sempre ser recuperados a partir do botão “Restaurar”.

Desvios das instruções RST

Este item leva à caixa de diálogo mostrada na figura 10.11.



Fig. 10.11 – Desvios das instruções RST

Da mesma forma como no caso das interrupções acima, as instruções RST desviam o processamento do programa para fora da região de memória simulada pelo Abacus. Para contornar este problema, esses desvios podem ser reprogramados. O programa se comporta então como se houvesse, naqueles endereços, instruções JMP para os novos endereços em RAM.

Os valores default podem ser sempre recuperados a partir do botão “Restaurar”.

10.4.4 O menu Ajuda

Este menu tem os seguintes itens:

Descrição do Abacus

Dá uma pequena descrição do programa e explica sua relação com este livro.

Sobre o Abacus

Tem informações sobre a versão do programa e a equipe que trabalhou no seu desenvolvimento.

10.5 Codificação de instruções do 8085

O primeiro passo no desenvolvimento de um programa deve ser sempre uma etapa de planejamento, cujo objetivo é esclarecer *o que* o programa deve fazer.

Somente depois deve-se passar ao detalhamento, etapa em que se decide *como* as coisas serão feitas. Esta etapa inclui a elaboração do código, que consiste em escolher as instruções a serem utilizadas e dispô-las em uma seqüência tal que produzam o resultado desejado.

Para o processador, cada instrução consiste de um ou mais números binários, que são lidos, interpretados e executados. No entanto, para os seres humanos, é difícil pensar em termos desses números. Por isso, os fabricantes de microprocessadores associam a cada instrução um *mnemônico*, que é uma espécie de abreviatura do que a instrução faz. Pode-se então utilizar os mnemônicos para escrever um programa, em vez de ter que lidar diretamente com os números binários.

Suponha, por exemplo, que um certo programa armazena na posição 2015H da memória um valor que desejamos incrementar. Poderíamos usar a seqüência:

```
LDA 2015H      ; Traz o conteúdo do byte 2015H para A
INR A          ; A = A + 1
STA 2015H      ; Escreve A no byte de endereço 2015H
```

Para que o processador possa “entender” o programa acima, as instruções devem ser traduzidas para o código binário que lhes corresponde. A tradução é feita de acordo com as especificações do fabricante, dadas pela *tabela de instruções* do microprocessador (v. anexo 1 e [INTE77]).

Os códigos correspondentes às instruções do exemplo, em hexadecimal, são:

```
3A 15 20      ; Codificação de LDA 2015H
3C             ; Idem, INR A
32 15 20      ; Idem, STA 2015H
```

É interessante notar que os bytes do endereço 2015H são codificados em ordem inversa (primeiro 15H, depois 20H). Esta é uma convenção seguida por todos os processadores da Intel: o armazenamento de valores de mais de um byte é feito de modo que o byte menos significativo fique no endereço menor e o byte mais significativo no maior.

Terminada a tradução, podemos tomar os códigos hexadecimais obtidos e colocá-los na memória do computador, para que o programa possa ser executado. Neste ponto, é necessário escolher o endereço a partir do qual esses bytes serão escritos na memória de programa. Por exemplo, se desejarmos que

o programa inicie na posição de memória 2000H, então os endereços ficarão assim:

2000H	3A 15 20	; Codificação de LDA 2015H
2003H	3C	; Idem, INR A
2004H	32 15 20	; Idem, STA 2015H

O programa terá ao todo 7 bytes e ocupará as posições de 2000H a 2006H, inclusive. É importante notar que não se deve chegar até a posição 2015H, pois esta é utilizada como a variável que está sendo incrementada.

Finalmente, a posição 2015H teria que ser inicializada com algum valor, para podermos testar o programa. Este valor poderia ser, por exemplo, 00H e, se depois de executar o programa, verificarmos novamente o valor dessa posição, deveríamos encontrar 01H.

Os opcodes acima podem ser colocados na memória do Abacus para executar o programa (veja o exercício 1 da seção 10.7).

10.6 O assembler do Abacus

No exemplo anterior, que tem apenas três instruções, é perfeitamente possível fazer a tradução dos mnemônicos para os opcodes manualmente. Também a inicialização da variável utilizada é uma tarefa simples, pois trata-se de apenas um byte. No entanto, para programas maiores, este método seria muito ineficiente.

O assembler do Abacus pode ajudar a resolver este problema, mas apenas se os programas forem pequenos. Embora este assembler permita criar, em princípio, qualquer programa que caiba na memória, ele tem limitações que tornam seu uso pouco interessante para desenvolver programas grandes. Um dos problemas é que, embora os programas criados no Abacus possam ser salvos em formato .hex, não existe uma listagem com comentários que possa ser impressa, guardada ou analisada. Isso tem um impacto inaceitável sobre a documentação, que é essencial em projetos maiores. Outra desvantagem importante é que não é possível dividir um programa em vários módulos, o que é essencial para que se possa trabalhar em equipe.

Para superar este tipo de limitação, trabalha-se, em projetos maiores, com ferramentas de software que dão suporte ao desenvolvimento de novos programas, como as apresentadas no capítulo 11. O assembler do Abacus deve ser visto, portanto, como uma ferramenta útil para estudar e fazer pequenos testes com agilidade, mas não como uma alternativa para os recursos descritos adiante.

10.7 Exercícios

1. Introduza os valores hexadecimais do programa da seção 10.5 na memória do Abacus, a partir do endereço 2000H. Execute as instruções passo a passo, acompanhando o que acontece no processador e na memória.
2. Inicie o Abacus e ative a janela Abacus Assembler, clicando no botão



“Exibir assembler”.

Introduza novamente o programa da seção 10.5, selecionando as instruções na janela de mnemônicos. Note que agora não é preciso recorrer à tabela de instruções para descobrir os opcodes de cada instrução. Esta tarefa foi resolvida pelo Abacus, e você pôde se concentrar apenas nos mnemônicos.

3. Modifique o programa anterior para que o conteúdo da posição de memória seja decrementado de uma unidade.
4. Escreva um programa que some os bytes 2015H e 2016H da memória e armazene o resultado dessa soma no endereço 2017H.

◊ Dica: leia o primeiro byte a ser somado para o acumulador, como nos exercícios anteriores. Depois guarde este valor num outro registrador, utilizando a instrução MOV (v. anexo 1 e [INTE77]). Em seguida, leia o segundo operando da soma, realize a operação com a instrução ADD e guarde o resultado.

5. O programa do exercício 4 só funciona corretamente se a soma não passar de FFH. Se houver um “vai 1”, este não é registrado. Corrija esta situação, alterando o programa de forma que o resultado passe a ocupar os bytes 2017H e 2018H. Lembre-se de que o byte mais significativo do resultado deve ser o de endereço mais alto e faça-o igual a zero se não houve “vai 1” e igual a 1 em caso contrário. Para descobrir se há “vai 1” ou não, veja a descrição das instruções JC e JNC.

11 Ferramentas de desenvolvimento

Este capítulo apresenta ferramentas simples de desenvolvimento de programas para o 8085, que podem ser utilizadas para criar os programas utilizados em todos os exemplos do livro para este processador. Mais importante, porém, do que aprender os detalhes de utilização dessas ferramentas, é compreender as razões da sua existência e o seu papel no processo de desenvolvimento, pois este conhecimento é de caráter geral e válido para todos os processadores.

11.1 O papel do assembler

Conforme exposto no capítulo 4, existem ferramentas de software, os assemblers, que fazem o trabalho de tradução de um arquivo-texto, contendo mnemônicos, para um arquivo-objeto, contendo opcodes. Para que isso seja possível, o arquivo-fonte tem que obedecer a certas convenções, que determinam a estrutura dos programas escritos em linguagem Assembly. O objetivo desta seção é apresentar alguns detalhes sobre essa estrutura e levar, com isso, a uma melhor compreensão do papel do assembler.

A listagem a seguir mostra como ficaria o programa do exemplo da seção 10.5, quando escrito como arquivo-fonte para o assembler do 8085.

```
ORG 2000H      ; Codificar a partir de 2000H
LDA COUNT     ; Codificação de LDA 2015H
INR A         ; Idem, INR A
STA COUNT     ; Idem, STA 2015H
ORG 2015H     ; Codificar a partir de 2015H
COUNT DB 00H ; Cria variável com valor inicial 00H
END
```

A primeira linha indica ao assembler que o código inicia no endereço 2000H. Com isso, o assembler tem como determinar os endereços de todas as instruções seguintes.

As linhas seguintes contêm os mnemônicos e geram o código correspondente a cada um deles. Mais adiante, vem uma segunda diretiva `ORG`, que diz ao assembler que tudo que for gerado dali em diante deve iniciar na posição 2015H. É o caso da diretiva

```
COUNT    DB 00H,
```

que diz ao assembler para reservar 1 byte no endereço de memória 2015H e estabelece, ao mesmo tempo, um sinônimo para este endereço, que pode ser utilizado no programa (como nas linhas `LDA COUNT` e `STA COUNT`).

O símbolo `COUNT`, no exemplo acima, é o que se chama de um *label* (etiqueta, rótulo). Os *labels* existem com a finalidade de poupar o programador do cálculo de endereços para colocar nas instruções. Por exemplo, se depois do byte `COUNT` fossem definidos outros, assim:

```
TABELA    DB 00H, 01H, 02H, 03H, 04H, 05H  
FLAG      DB 01H
```

então o símbolo `TABELA` seria um sinônimo para os valores 2016H, que é o endereço do primeiro byte da tabela. Conseqüentemente, os demais bytes da tabela ocupariam as posições 2017H a 201BH, e o símbolo `FLAG` seria um sinônimo para o valor 201CH. Isso é muito útil, principalmente nos casos em que se decide mudar alguma coisa de lugar no programa. Se, ao invés de colocar as variáveis a partir do endereço 2015H, como no exemplo acima, decidíssemos passá-las para um outro lugar (mudando o argumento da diretiva `ORG`), então os *labels* seriam atualizados automaticamente pelo assembler. Caso usássemos os endereços diretamente dentro das instruções, todo o programa teria que ser revisado para se adaptar aos novos endereços. O mesmo vale para o caso de se decidir mudar o comprimento da tabela.

Os valores colocados após a diretiva `DB` são inicializam as variáveis. Esses valores são colocados no arquivo-objeto e, quando o programa é carregado para execução, são transferidos para a memória juntamente com os códigos das instruções. No caso do exemplo acima, isto poupa o programador de ter que inicializar a variável `COUNT` antes de testar o programa. Note, porém, que as variáveis em questão só são inicializadas cada vez que o programa é transferido do disco para a memória. Se o programa for executado até um ponto em que essas variáveis tenham sido alteradas e então executado novamente do início, os valores das variáveis não corresponderão mais aos da inicialização. Por isso, ao escrever programas que não sejam apenas exemplos didáticos, é boa prática incluir, no início, instruções para inicializar todas as variáveis.

Qualquer linha, inclusive as de código, pode iniciar por um label, que serve então para que se possa referenciá-la em outros pontos do programa, como no trecho de código seguinte:

```
                MVI B,00H
LOOP:          ADD B
                DCR C
                JNZ LOOP
```

Este trecho de código pode servir para fazer a multiplicação do número contido no acumulador pelo número contido no registrador C. O resultado é obtido em B, que recebe C vezes o número contido em A. Para isto, o loop formado pelas três últimas instruções tem que ser repetido C vezes, e a instrução JNZ LOOP desvia o processamento do programa de volta para a instrução ADD B enquanto o registrador C não chegar a zero. Para poder fazer este desvio, a instrução precisa receber como argumento o endereço dessa instrução. O uso do label LOOP dispensa o programador de se preocupar com o valor do endereço.

Finalmente, o caracter ‘;’ (ponto e vírgula) serve para iniciar um comentário, que se encerra automaticamente no final da linha. Qualquer texto escrito como comentário não tem influência sobre o código gerado e serve apenas como documentação. O uso de comentários deve ser feito com cuidado, para evitar a poluição visual da listagem. Em geral, um comentário não deve dizer *o que* a instrução faz, porque isso é uma coisa que normalmente o programador já sabe ou pode descobrir mediante consulta à tabela de instruções, mas sim *para que* ela está servindo no ponto do programa onde se encontra. Por exemplo, escrever:

```
        LDA 2015H ; carrega acumulador com conteúdo da
                  ; posição 2015H da memória
```

é diferente de

```
        LDA 2015H ; lê contagem de peças na esteira
```

Uma exceção para esta regra são os primeiros programas, em que comentários como o primeiro acima (e os da seção 10.5) podem ajudar a fixar alguns conhecimentos.

11.2 Utilização das ferramentas de software

Esta seção apresenta as ferramentas utilizadas nas diversas etapas do desenvolvimento de um programa.

11.2.1 Edição

Nesta etapa, pode-se utilizar qualquer editor que permita gravar arquivos em formato texto (ASCII). É importante lembrar que os editores que permitem acrescentar formatação ao texto geralmente armazenam esta informação na forma de caracteres não visíveis, que atrapalham o assembler na hora de ler o arquivo. Por isso, é importante ter certeza de que o arquivo é salvo realmente apenas como texto.

Os labels devem sempre iniciar na primeira coluna do arquivo-fonte e as instruções e diretivas nunca na primeira coluna. Recomenda-se colocá-las a partir da coluna 9. Para tanto, convém configurar o editor para que um TAB corresponda a 8 espaços.

O arquivo deve ter um nome de até oito letras e extensão .asm (por exemplo Lab01.asm).

11.2.2 Montagem

O assembler utilizado neste texto é o X8085, um programa para o sistema operacional MS-DOS que usa o arquivo .asm como entrada e gera os arquivos .lst e .obj, conforme mostra a figura 11.1.

O arquivo com extensão .obj contém os códigos hexadecimais e se destina à geração do arquivo que será lido pelo processador, enquanto o arquivo .lst é uma listagem destinada ao programador. Esta contém informações importantes tais como os endereços em que foram colocadas as instruções, valores dos símbolos e códigos hexadecimais gerados.

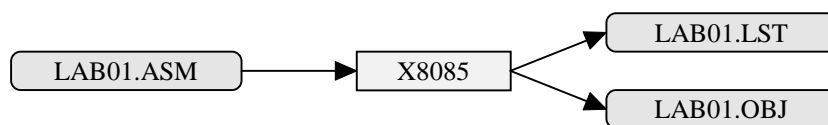


Fig. 11.1 – O processo de montagem (assembly)

O X8085 pode ser chamado de duas formas. A primeira, mais extensa, solicita os parâmetros necessários, de acordo com o que segue:

```
C:\EEL7030>x8085
```

```
8085 CROSS ASSEMBLER - COPYRIGHT 1984 BY 2500 A.D. SOFTWARE, INC
```

```
VERSION 3.41a SERIAL # 040-051-0105
```

```
LISTING DESTINATION ? (TI,LP,DD,NL,DL,EO):
```

Aqui existem as seguintes opções:

- TI (Terminal) - mostra listagem na tela
- DD (Disk Drive) - gera listagem num arquivo em disco
- LP (Line Printer) - gera listagem na impressora
- NL (No Listing) - sem listagem (opção default)
- DL (Directive Listing) - lista apenas as diretivas do assembler
- EO (Error Only) - gera listagem apenas se houver erros

```
LISTING DRIVE ? (<CR> = SAME AS OUTPUT FILE)
```

Obs.: esta pergunta não aparece se não foi solicitada a listagem em disco.

```
GENERATE CROSS REFERENCE ? (Y/N, <CR> = NO)
```

Gera lista de referências cruzadas, útil para depurar programas grandes.

```
INPUT FILENAME ? : <nome do arquivo, sem a extensão .asm> <Enter>
```

```
OUTPUT FILENAME ? : <Enter>
```

A segunda forma de chamar o X8085 é passar-lhe os parâmetros na linha de comando, por exemplo com:

```
C:\EEL7030>X8085 DD,,,LAB01,,
```

que diz que a o X8085 deve procurar o arquivo Lab01.asm no diretório atual e gerar, a partir deste, o arquivo-objeto Lab01.obj e a listagem Lab01.lst. Esta deve ser colocada no disco, no mesmo diretório onde se encontra o programa-fonte.

Em qualquer dos casos, se houver erros de montagem, é necessário voltar à etapa de edição e corrigir o programa.

11.2.3 Linkagem

O código objeto precisa ser colocado num formato que possa ser lido pelo simulador. Para isto utiliza-se um linker, no nosso caso o LINK2, que toma um ou mais arquivos .obj como entrada e gera um arquivo .hex:



Fig. 11.2 – O processo de linkagem

O arquivo .hex é do tipo texto, e seu formato é ilustrado de acordo com o exemplo a seguir:

:08	2000	00	3A15203C32152076	50	
				Checksum	
				Código do programa	
				00, exceto na última linha	
				Endereço de carga na memória	
				Número de bytes de programa	
				contidos nesta linha (<= 10H)	
:00000001FF					Linha final do arquivo

Ao contrário do X8085, o LINK2 não pode ser chamado com parâmetros na linha de comando. Os dados de entrada do LINK2, para o caso de um único arquivo-objeto, são:

INPUT FILENAME: <nome do arquivo, sem a extensão .obj>

LOAD ADDRESS (OFFSET): <Enter>

INPUT FILENAME: <Enter>

OUTPUT FILENAME: <Enter>

11.2.4 Execução

Os arquivos .hex gerados pelo LINK2 podem ser lidos e executados no Abacus, conforme descrito no capítulo 10.

11.3 Exemplo

Seja a tarefa de desenvolver um programa que inicie lendo, da posição de memória 2022H, um número inteiro N . Os N bytes armazenados a partir da posição 2023H devem então ser somados e o resultado da soma, que pode exceder FFH, deve ser colocado nos endereços 2020H e 2021H, seguindo a convenção da Intel de armazenar o byte menos significativo no endereço mais baixo. Deve-se ainda criar dados de teste no final do programa, para facilitar sua verificação.

Um início de solução, ainda com problemas, é dado a seguir.


```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Soma.asm - soma de numeros em hexadecimal
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ORG 2000H
        LDA N                ; traz para A o valor de N
        MOV C,A              ; guarda este valor em C
        LXI H,DADOS          ; aponta par HL p/ inicio da tabela
        MVI B,00H            ; inicializa soma
LOOP:    MOV A,M              ; le dado apontado por HL
        ADD B                ; soma acumulada em A
        MOV B,A              ; e guardada em B
        INX H                ; par HL aponta para o dado seguinte
        DCR C                ; um a menos para fazer
        JNZ LOOP            ; repete ate que C = 0
        LXI H,RESULT         ; aponta HL p/ inicio do resultado
        MOV M,B              ; guarda B na memoria
        JMP $                ; loop infinito, fica parado aqui
        ORG 2020H
RESULT  DB 00H,00H
N       DB 03H
DADOS   DB 01H,02H,03H,04H,65H,76H,87H,98H,A9H
        END

```

Note que este programa ainda apresenta os seguintes problemas:

- se $N=0$, são somados 256 valores;
- o resultado está sendo calculado em apenas 1 byte e fica incorreto se passar de FFH;
- se aumentarmos o tamanho do programa, corremos o risco de usar os endereços de memória destinados aos dados.

11.4 Exercícios

1. Monte e teste o programa apresentado no simulador, certificando-se de que funciona. Altere o valor inicial de N e verifique as mudanças no comportamento do programa.
2. Modifique-o de modo que não faça a soma caso N seja nulo.
 - ◊ Dica: estude as instruções CPI e JZ em [INTE77]. Use a instrução CPI após ler N para o acumulador, a fim de determinar se seu valor é zero, e coloque uma instrução de desvio condicional (JZ) para desviar para o final do programa caso seja. Para fazer o desvio, use um label na linha para a qual quer desviar, de forma análoga ao caso da instrução JNZ LOOP.

3. Modifique o programa de modo que os dois bytes previstos para o armazenamento do resultado sejam utilizados.

◊ Dica 1: estude as instruções JC e JNC em [INTE77]. Utilize uma delas para determinar se houve “vai 1” na soma e, caso tenha havido, incremente um outro registrador (previamente inicializado em zero) para armazenar a parte mais significativa da soma. No final, armazene este registrador no endereço RESULT+1:

```
LOOP:    ...           ; Inicializa regs para armazenar a soma
          MOV A,M
          ADD B
          MOV B,A       ; MOV não afeta os flags
          JNC OK        ; Se não houve “vai 1”, está tudo bem
          ...           ; Instrução a executar se houve “vai 1”
OK:      DCR C
          JNZ LOOP
          LXI H,RESULT
          MOV M,B
          INX H         ; Aponta para o byte mais significativo
          ...           ; Armazena o byte mais significativo da
                        ; soma
```

◊ Dica 2: note que, depois das modificações, o programa pode ter aumentado de forma a ocupar os endereços 2020H e seguintes. Se isso acontecer, não será mais possível guardar os dados a partir do endereço 2020H, pois isso destruiria o próprio programa. Você pode verificar como as coisas estão, analisando os endereços no arquivo .lst gerado pelo X8085. Experimente retirar a diretiva ORG 2020H do programa. Isso fará com que os dados iniciem imediatamente após o código. Note que, graças ao uso dos labels, não é preciso modificar o programa-fonte em ponto algum.

12 Sub-rotinas simuladas em ROM

O trabalho com pequenos computadores, tal como o simulado pelo Abacus, envolve tarefas como a leitura do teclado e a apresentação de resultados no display. Considerando a elevada frequência com que tais tarefas precisam ser executadas, esse tipo de computador didático normalmente inclui algumas sub-rotinas de serviço que facilitam o acesso ao hardware, armazenadas em memória ROM. O mesmo vale para o Abacus, que simula a presença desse tipo de sub-rotina. O objetivo deste capítulo é apresentar essas sub-rotinas e ilustrar sua utilização. Recomenda-se a leitura das seções 3.3 e 3.4 do capítulo sobre o 8085 para uma melhor compreensão do material que segue.

12.1 As sub-rotinas

A tabela 12.1 apresenta as sub-rotinas do Abacus, seus endereços e os registradores que são alterados quando são chamadas.

Nome	Endereço	Finalidade	Regs.
MOSTRAA	036EH	Mostrar A no campo de dados	Todos
MOSTRAD	0363H	Par DE no campo de endereços	Todos
LETECLA	02E7H	Tecla 0...F no acumulador	A,H e L
DELAY	05F1H	Espera proporcional a D	A,D e E

Tab. 12.1 – Sub-rotinas simuladas em ROM

12.2 Exemplo

Ler, a partir do teclado, dois números hexadecimais de um dígito. Exibir estes números no campo de endereços e sua soma no campo de dados.

Observação: a leitura do teclado do kit Telemática, no qual foi baseado o Abacus, utiliza a interrupção RST 5.5. Programas executados neste kit precisam, por isso, habilitar essa interrupção para que o teclado funcione, incluindo as linhas

```
MVI A,18H
SIM
```

no início do programa, conforme as explicações do capítulo 5. As listagens apresentadas neste livro não incluem estas linhas, porque no Abacus a leitura do teclado é independente das interrupções.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Teclado.asm - leitura de numeros do teclado
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
LETECLA EQU 02E7H      ; Sinonimos utilizados abaixo
MOSTRAD EQU 0363H      ;
MOSTRAA EQU 036EH      ;

                ORG 2000H

                LXI SP,20C0H      ; Inicializa pilha
LOOP:          CALL LETECLA      ; Le primeiro numero
                MOV D,A
                MVI E,0H
                PUSH D
                CALL MOSTRAD      ; Apresenta primeiro numero
                POP D
                CALL LETECLA      ; Le segundo numero
                MOV E,A
                ADD D              ; Resultado em A
                PUSH PSW
                CALL MOSTRAD      ; Apresenta os dois numeros
                POP PSW
                CALL MOSTRAA      ; Apresenta resultado
                JMP LOOP
                END
```

12.3 Exercícios

1. Modifique o exemplo acima para permitir a entrada de números hexadecimais de dois dígitos.

◊ Dica: estude a instrução RLC em [INTE77]. Note que, se o acumulador contiver um dígito entre 0 e F e for rotacionado com RLC, isto equivale a multiplicar o dígito por 2. Portanto, fazendo esta operação 4 vezes, o conteúdo do acumulador será multiplicado por $2^4 = 16$. Em hexadecimal, isto equivale a acrescentar um zero ao final do número: por exemplo, 07H torna-se 70H. Isto permite ler números de dois dígitos:

```
CALL LETECLA      ; Lê dígito mais significativo
RLC
RLC
RLC
RLC               ; Multiplica por 16
MOV D,A           ; e guarda isso em D
CALL LETECLA      ; Lê dígito menos significativo
ADD D             ; Número de dois dígitos em A
```

2. Modifique o exemplo acima de modo a se ter um somador decimal, evitando, inclusive, que o usuário entre com dígitos hexadecimais (*n* não deve ser maior do que 9).

◊ Dica 1: estude as instruções CPI e JP em [INTE77] e utilize-as para evitar a entrada de dígitos superiores a 9:

```
LOOP1: CALL LETECLA      ; Lê dígito mais significativo
CPI 0AH
JP LOOP1           ; Volta a ler dígito se era > 9
```

◊ Dica 2: estude a instrução DAA em [INTE77]. Esta implementa o algoritmo de correção da soma de dois números BCD. Note que os números que você está armazenando agora são números BCD, pois os dígitos A-F nunca aparecem. Para fazer a soma de dois números BCD, basta proceder como no caso anterior e corrigir o resultado com DAA:

```
...               ; primeiro número (2 dígitos) -> D
...               ; 1º dígito do 2º número, x 16 -> E
CALL LETECLA      ; 2º dígito do 2º número em A
ADD E             ; 2º número de dois dígitos em A
ADD D             ; soma em A, ainda sem correção
DAA               ; A contém soma BCD corrigida
```

3. Escreva um programa que receba dois números hexadecimais a e b pelo teclado e efetue a operação $y = 2a - b$. O valor de y deve ser mostrado no campo de endereços.
 - ◇ Dica: utilize novamente a instrução RLC (v. exercício 1) e estude a instrução SUB.
4. Repita o exercício anterior para números decimais.
 - ◇ Dica: trabalhe com os números em BCD e utilize novamente a instrução RLC para fazer a multiplicação por 2. Agora não é possível usar SUB, porque não existe uma instrução equivalente ao DAA para a subtração. Em vez disso, faça a subtração através da soma, conforme explicado na seção 2.4.
5. Escreva um programa que receba um número a entre 0 e 7, inclusive, pelo teclado e efetue a operação $y = 2^a$. O valor de y deve ser mostrado no campo de dados.
 - ◇ Dica: utilize novamente a instrução RLC para rotacionar o acumulador a vezes.
6. Escreva um programa que receba um número hexadecimal de dois dígitos pelo teclado e que conte o número de bits "1" contidos neste número. O número digitado deve ser mostrado no campo de dados do mostrador e o resultado da contagem, no mostrador de endereços.
 - ◇ Dica: utilize novamente a instrução RLC para provocar a passagem de cada um dos 8 bits pelo carry flag. À medida que os bits vão passando, utilize uma das instruções JC ou JNC para determinar se a contagem de '1's deve ser incrementada ou não.
7. Escreva uma sub-rotina que faça a divisão de um número de 16 bits colocado no par DE por um número de 8 bits colocado no registrador C. Após o retorno, o quociente da divisão deve estar no par DE e o resto no registrador L. A sub-rotina não deve alterar o registrador B. Utilize esta sub-rotina num programa que leia dois números do teclado, faça sua divisão e apresente o resultado.
 - ◇ Dica 1: dividir X por Y significa calcular quantas vezes Y cabe em X. Portanto, o quociente da divisão representa o número de vezes que é possível subtrair Y de X sem que o resultado seja negativo.
 - ◇ Dica 2: a solução deste exercício é parte do programa da seção 14.1. Consulte-o se achar necessário, mas procure resolver o problema sem ajuda antes.

8. Escreva um programa que aceite dígitos hexadecimais do teclado e os mostre no campo de endereços do mostrador. Os dígitos devem ser mostrados um a um, passando da direita para esquerda a cada digitação, como no exemplo a seguir.

Campo de endereços	Dígitos teclados
[][][][3]	3
[][][3][F]	F
[][3][F][D]	D
[3][F][D][0]	0
[F][D][0][4]	4

- ◇ Dica 1: estude a instrução RAL e utilize o carry flag para transportar os bits de um registrador a outro. A passagem do bit mais significativo do registrador E para o bit menos significativo do registrador D, por exemplo, pode ser feita com:

```
MOV A,E
RAL
MOV E,A
MOV A,D
RAL
MOV D,A
```

- ◇ Dica 2: utilize a instrução ORA A para zerar o carry flag quando precisar.

13 Data juliana

13.1 Objetivo

Os exercícios deste capítulo têm por objetivo a fixação dos seguintes conhecimentos:

- endereçamento indireto da memória;
- utilização de laços de repetição;
- percepção da necessidade de utilizar números hexadecimais e BCD.

13.2 Exercício proposto

Em 1583, o francês Julius Joseph Scaliger (1540-1609) introduziu o assim chamado calendário juliano, que não conta meses e anos, mas apenas os dias que se passaram desde o meio-dia do dia 1.º de janeiro de 4713 a.C. Assim, por exemplo, o dia 1.º de janeiro de 1982 é o dia 2440970,5. As datas julianas são úteis para registrar eventos quando é necessário calcular o número de dias que os separam, bastando para isso uma simples subtração.

Mais recentemente, introduziu-se também a data juliana modificada, que difere da anterior apenas pela escolha da origem, que é a meia-noite do dia 17 de novembro de 1858.

Para este exercício, vamos definir um novo tipo de data juliana, cuja origem é o dia 1.º de janeiro do ano em que estivermos, e sem considerar anos bissextos, para simplificar. O dia 1.º de junho, por exemplo, é o 152.º dia do ano e, por isso, a data juliana correspondente a esse dia é 152.

Sua tarefa é implementar um conversor de datas da forma dd/mm para a nova data juliana. Uma vez iniciado, o programa deve ser executado de acordo com os seguintes passos:

1. o usuário deve poder digitar dois números de dois dígitos cada, correspondendo ao dia (dd) e ao mês (mm), nesta ordem;
 - ◊ não é necessário verificar se a data é válida (por exemplo, evitar que se digite um número maior do que 30 para um dia do mês de junho); basta evitar que o usuário entre com dígitos hexadecimais;
 2. assim que for introduzido o segundo dígito correspondente ao mês, o programa deverá iniciar o cálculo da data juliana correspondente;
 - ◊ dica: utilize uma tabela com os números de dias de cada mês do ano; procure avaliar as vantagens e desvantagens de utilizar valores decimais ou hexadecimais (BCD) nessa tabela;
 3. o resultado deverá aparecer em forma decimal (um número entre 1 e 365, inclusive) no display de endereços;
 4. o programa deverá então voltar ao passo 1, para que se possa converter outra data.
- ✎ A seção 13.3 contém uma solução para este exercício. Procure consultá-la somente depois que tiver chegado a um programa seu!

13.3 Solução

```
;;;;;;;;;;;;;
;Juliana.asm - conversor de datas
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;
LETECLA EQU 02E7H
MOSTRAD EQU 0363H

        ORG 2000H
        LXI SP,20C0H

LE1:     CALL LETECLA      ; 1. digito do dia
        CPI 0AH
        JP LE1
        RLC
        RLC
        RLC
        RLC
        MOV E,A
LE2:     CALL LETECLA      ; 2. digito do dia
        CPI 0AH
        JP LE2
        ORA E
        MOV E,A
        MVI D,00H        ; E contem dia em BCD e D = 0
LE3:     CALL LETECLA      ; 1. digito do mes
        CPI 0AH
        JP LE3
        MVI C,00H
        CPI 00
        JZ LE4
        MVI C,10D
LE4:     CALL LETECLA      ; 2. digito do mes
        CPI 0AH
        JP LE4
        ADD C
        SUI 01H
        MOV C,A          ; C = mes-1 = numero de meses a somar
        CPI 00H
        JZ MOSTRA        ; se mes = janeiro, esta pronto
        LXI H,DIAS
        MOV A,E           ; A inicia com o dia...
SOMA:    ADD M             ; ... e recebe os dias de cada mes
        DAA
        JNC CONT
        INR D             ; "vai 1"
CONT:    INX H
        DCR C
        JNZ SOMA
        MOV E,A
MOSTRA:  CALL MOSTRAD      ; resultado no campo de enderecos
```

```
JMP LE1          ; volta ao passo 1 do enunciado
DIAS DB 31H,28H,31H,30H,31H,30H
      DB 31H,31H,30H,31H,30H,31H
      END
```

13.4 Exercícios complementares

Os exercícios abaixo exploram a utilização do endereçamento indireto no trabalho com tabelas.

1. Escreva um programa que efetue a varredura do bloco de memória entre 20A0H e 20AFH e indique a quantidade de números ímpares presentes no bloco. Crie uma tabela para inicializar o conteúdo do bloco, de maneira a facilitar os testes. Opcionalmente, complemente o programa de modo que seja possível digitar, numa primeira etapa, dados para armazenar na tabela e depois contar quantos dados eram ímpares.
 - ♦ Dica: um número é ímpar quando seu bit menos significativo é igual a 1. Isso lembra manipulação de bits (v. seção 2.6).
2. Escreva um programa que transfira, em ordem inversa, os conteúdos das posições de memória do bloco entre 20A0H e 20AFH para o bloco entre 2080H e 208FH.
 - ♦ Dica: pode ser interessante estudar as instruções XCHG, LDAX e STAX.
3. Escreva um programa que permita entrar com um número via teclado e procurar por este valor na região de memória entre 20B0H e 20BFH. Se encontrado, o endereço no qual este se encontrava deve aparecer no campo de endereços. Caso contrário, o campo de endereços deve apresentar o conteúdo FFFFH. Em ambos os casos, o número procurado deve ser mostrado no campo de dados do mostrador. Para facilitar os testes, crie uma tabela com valores conhecidos na região 20B0H a 20BFH.
4. Escreva um programa que mostre a si mesmo. Os endereços dos bytes que o compõem devem aparecer no campo de endereços e seus códigos hexadecimais no campo de dados. Depois que o programa tiver se mostrado por inteiro, deve recomeçar do primeiro byte.

14 Conversão de base

O objetivo deste capítulo é a análise dois programas que implementam o algoritmo de conversão de base apresentado no capítulo 2. Este tipo de conversão é muito útil em programas que precisam apresentar resultados em sistemas de numeração diferentes do sistema hexadecimal. Revise a teoria antes de continuar, se achar necessário.

O primeiro programa, apresentado na seção 14.1, é denominado HexConv.asm. Recebe do teclado um número hexadecimal de 4 dígitos e uma base de saída (um número decimal de dois dígitos). Esta deve estar entre 2 e 16; do contrário os resultados do programa não fazem sentido.

O segundo programa, cuja listagem inicia na página 177, se distingue do primeiro por permitir a conversão entre números em quaisquer bases entre 2 e 16. Assim, recebe do teclado primeiramente uma base de entrada (um número decimal de dois dígitos), depois um número de quatro dígitos a ser convertido e, por último, uma base de saída (novamente um número decimal de dois dígitos).

Ambos os programas iniciam o cálculo do resultado assim que termina a entrada dos dados e o apresentam no display. A estratégia adotada para a apresentação do resultado também é interessante: para mostrar o maior número possível de dígitos, os campos de endereços e de dados foram considerados como um único número de seis algarismos. Estes campos são preenchidos fazendo com que os dígitos entrem pela direita (parte menos significativa do campo de dados) e se desloquem para a esquerda à medida que mais dígitos são introduzidos. É interessante que você tenha resolvido os exercícios indicados na seção 12.3, antes de estudar esta parte dos programas.

14.1 Listagem de HexConv.asm

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;HexConv.asm - conversao de base para numeros hexadecimais
;Recebe do teclado um numero hexadecimal de 4 digitos
;Recebe a base de saida (um numero decimal de 2 digitos)
;A base de saida deve estar entre 2 e 16, inclusive
;Converte o numero para essa base; resultado no display
;Campos de enderecos e dados formam um numero de 6 digitos
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
LETECLA EQU 02E7H
MOSTRAD EQU 0363H
MOSTRAA EQU 036EH

                ORG 2000H
                LXI SP,2100H
LOOP:          CALL LENUM16 ; DOIS DIGITOS MAIS SIGNIFICATIVOS
                MOV D,A
                CALL LENUM16 ; DOIS DIGITOS MENOS SIGNIFICATIVOS
                MOV E,A
                PUSH D
                CALL MOSTRAD ; APRESENTA NUMERO A SER CONVERTIDO
                POP D
                CALL LENUM10 ; BASE DE SAIDA
                MOV C,A
                CALL HEXCONV ; CONVERTE. RESULTADO EM D, E e A
                PUSH PSW
                CALL MOSTRAD
                POP PSW
                CALL MOSTRAA
                JMP LOOP

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;HEXCONV - converte numero hexa para a base de saida
;Recebe numero no par DE e base de saida em C
;Retorna resultado como numero de 6 digitos em D, E e A
;Utiliza todos os regs
;Chama a sub-rotina DIV
HEXCONV:
                MVI B,00H          ; CONTADOR DE DIGITOS DO RESULTADO
MORE:          CALL DIV            ; QUOCIENTE EM DE E RESTO EM L
                PUSH H              ; L CONTEM DIGITO DO RESULTADO
                INR B                ; CONTA DIGITOS
                MOV A,D
                CPI 00H              ; SE D <> 0, TEM MAIS
                JNZ MORE
                MOV A,E
                CPI 00H              ; SE D == 0 MAS E <> 0, TEM MAIS
                JNZ MORE

```

```

    MVI A,00H      ; TRIPLA DEA CONTEM 0 PARA COMECAR
GETDIG: MVI H,04H   ; RESGATE DOS DIGITOS DA PILHA
ROTATE: ORA A      ; ZERA CARRY FLAG
        RAL        ; BIT MAIS SIGNIFICATIVO DE A EM CY
        MOV L,A
        MOV A,E     ; ROTACIONA DIGITOS DA TRIPLA DEA
        RAL        ; BIT MAIS SIGNIFICATIVO DE A EM E
        MOV E,A
        MOV A,D
        RAL        ; BIT MAIS SIGNIFICATIVO DE E EM D
        MOV D,A
        MOV A,L
        DCR H
        JNZ ROTATE
        POP H       ; L CONTEM DIGITO PARA A TRIPLA DEA
        ORA L       ; INTRODUZ DIGITO RESGATADO DA PILHA
        DCR B       ; TEM MAIS DIGITOS?
        JNZ GETDIG
        RET        ; NUMERO CONVERTIDO EM DEA

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;DIV - sub-rotina para divisao
;Recebe dividendo em DE e divisor em C
;Retorna quociente em DE e resto em L
;Utiliza regs A, C, D, E, H e L
DIV:   LXI H,0000H
        MOV A,E
TEST1:  CMP C
        JC TEST2
BACK:   SUB C
        INX H
        JMP TEST1
TEST2:  MOV E,A
        MOV A,D
        CPI 00H
        JZ DONE
        DCR D
        MOV A,E
        JMP BACK
DONE:   XCHG
        RET
```

```
;;;;;;;;;;;;;  
;LENUM10 - le numero decimal de dois digitos  
;Retorna valor hexa do numero lido no acumulador  
;Utiliza os regs A, B, H e L  
;Chama LETECLA  
LENUM10:
```

```
    CALL LETECLA  
    RLC  
    MOV B,A  
    RLC  
    RLC  
    ADD B  
    MOV B,A  
    CALL LETECLA  
    ADD B  
    RET
```

```
;;;;;;;;;;;;;  
;LENUM16 - le um numero hexadecimal de dois digitos  
;Retorna valor do numero lido no acumulador  
;Utiliza os regs A, B, H e L  
;Chama LETECLA  
LENUM16:
```

```
    CALL LETECLA  
    RLC  
    RLC  
    RLC  
    RLC  
    MOV B,A  
    CALL LETECLA  
    ADD B  
    RET  
END
```


14.2 Listagem de BaseConv.asm

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;BaseConv.asm - programa de conversao de base
;Recebe a base de entrada (numero decimal de 2 digitos)
;Recebe um numero de 4 digitos nessa base, sem verifica-lo
;Recebe a base de saida (um numero decimal de 2 digitos)
;Ambas as bases devem estar entre 2 e 16, inclusive
;Converte o numero da base de entrada para a base de saida
;Apresenta o resultado no display
;Campos de enderecos e dados formam um numero de 6 digitos
;Prof. Roberto m. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
LETECLA EQU 02E7H
MOSTRAD EQU 0363H
MOSTRAA EQU 036EH

                ORG 2000H
                LXI SP,2100H
LOOP:          CALL LENUM10 ; BASE DE ENTRADA LIDA (DECIMAL)
                MOV C,A
                CALL LENUM   ; LE NUMERO DE 4 DIGS, CONVERTE P/ HEXA
                PUSH D
                CALL MOSTRAD ; APRESENTA NUMERO A CONVERTER EM HEXA
                POP D
                CALL LENUM10 ; BASE DE SAIDA
                MOV C,A
                CALL HEXCONV ; CONVERTE. RESULTADO EM D, E e A
                PUSH PSW
                CALL MOSTRAD
                POP PSW
                CALL MOSTRAA
                JMP LOOP

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;HEXCONV - converte numero hexa para a base de saida
;Recebe numero no par DE e base de saida em C
;Retorna resultado como numero de 6 digitos em D, E e A
;Utiliza todos os regs
;Chama a sub-rotina DIV
HEXCONV:
                MVI B,00H          ; CONTADOR DE DIGITOS DO RESULTADO
MORE:          CALL DIV            ; QUOCIENTE EM DE E RESTO EM L
                PUSH H             ; L CONTEM DIGITO DO RESULTADO
                INR B              ; CONTA DIGITOS
                MOV A,D
                CPI 00H            ; SE D <> 0, TEM MAIS
                JNZ MORE
                MOV A,E
                CPI 00H            ; SE D == 0 MAS E <> 0, TEM MAIS
                JNZ MORE

```

```
MVI A,00H          ; TRIPLA DEA CONTEM 0 PARA COMECAR
GETDIG: MVI H,04H    ; RESGATE DOS DIGITOS DA PILHA
ROTATE: ORA A        ; ZERA CARRY FLAG
          RAL         ; BIT MAIS SIGNIFICATIVO DE A EM CY
          MOV L,A
          MOV A,E      ; ROTACIONA DIGITOS DA TRIPLA DEA
          RAL         ; BIT MAIS SIGNIFICATIVO DE A EM E
          MOV E,A
          MOV A,D
          RAL         ; BIT MAIS SIGNIFICATIVO DE E EM D
          MOV D,A
          MOV A,L
          DCR H
          JNZ ROTATE
          POP H        ; L CONTEM DIGITO PARA A TRIPLA DEA
          ORA L        ; INTRODUZ DIGITO RESGATADO DA PILHA
          DCR B        ; TEM MAIS DIGITOS?
          JNZ GETDIG
          RET          ; NUMERO CONVERTIDO EM DEA

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;DIV - sub-rotina para divisao
;Recebe dividendo em DE e divisor em C
;Retorna quociente em DE e resto em L
;Utiliza regs A, C, D, E, H e L
DIV:      LXI H,0000H
          MOV A,E
TEST1:    CMP C
          JC TEST2
BACK:     SUB C
          INX H
          JMP TEST1
TEST2:    MOV E,A
          MOV A,D
          CPI 00H
          JZ DONE
          DCR D
          MOV A,E
          JMP BACK
DONE:     XCHG
          RET
```

```
;;;;;;;;;;;;;
;MULT - sub-rotina para multiplicacao
;Recebe multiplicando em DE e multiplicador em C
;Retorna resultado em DE, preservando o multiplicador em C
;Utiliza regs A, C, D, E, H e L
MULT:  XCHG
        LXI D,0000H
        MOV A,C
        CPI 00H
        RZ
        PUSH B
MUL:   MOV A,E
        ADD L
        MOV E,A
        MOV A,D
        ADC H
        MOV D,A
        DCR C
        JNZ MUL
        POP B
        RET

;;;;;;;;;;;;;
;LENUM
;Le numero de 4 digitos na base contida em C
;Converte p/ hexa e retorna esse valor no par DE
;Utiliza todos os regs
;Chama MULT
LENUM:  LXI D,0000H
        MVI B,04H
MULBAS: CALL MULT      ; MULTIPLICA PAR DE PELA BASE EM C
        CALL LETECLA
        ADD E
        MOV E,A
        JNC CONT
        INR D
CONT:   DCR B
        JNZ MULBAS
        RET
```

```
////////////////////////////////////  
;LENUM10 - le um numero decimal de dois digitos  
;Retorna valor hexa do numero lido no acumulador  
;Utiliza os regs A, B, H e L  
;Chama LETECLA  
LENUM10:  
    CALL LETECLA  
    RLC  
    MOV B,A  
    RLC  
    RLC  
    ADD B  
    MOV B,A  
    CALL LETECLA  
    ADD B  
    RET  
    END
```

14.3 Exercícios complementares

Os exercícios abaixo envolvem tomada de decisões, conforme os princípios apresentados na seção 4.8.1. Embora não tenham relação com o assunto principal do capítulo, foram colocados aqui porque seu nível de dificuldade é semelhante ao dos programas de conversão de base.

1. Escreva um programa que receba três números hexadecimais de dois dígitos pelo teclado e mostre no campo de dados o maior dentre eles. Os outros dois números devem ser mostrados no campo de endereços do mostrador, o menor à esquerda e o do meio à direita.
2. Escreva um programa que receba três números hexadecimais de dois dígitos pelo teclado e mostre no campo de dados o menor dentre eles. Os outros dois números devem ser mostrados no campo de endereços do mostrador, o maior à esquerda e o do meio à direita.
3. Escreva um programa que coloque em ordem crescente os dados contidos na região de 20B0H a 20BFH. Depois de fazer a ordenação, o programa deve entrar em um loop que mostre os endereços e dados no mostrador, um de cada vez, voltando ao endereço 20B0H após mostrar a tabela toda. O avanço de um endereço para o seguinte deve acontecer a cada toque de qualquer das teclas de 0 a F.
 ♦ Dica: crie uma tabela com dados conhecidos no final do programa, para facilitar os testes.
4. Repita o exercício anterior, mas colocando os dados em ordem decrescente.

15 Interrupções na prática

15.1 Utilização das teclas de interrupção do Abacus

Para a utilização das interrupções no Abacus, é preciso levar em conta as seguintes características:

- as teclas RST 5.5, RST 6.5, RST 7.5 e Trap se comportam como se estivessem ligadas ao microprocessador 8085 da forma ilustrada na figura 15.1 para a tecla 7.5;

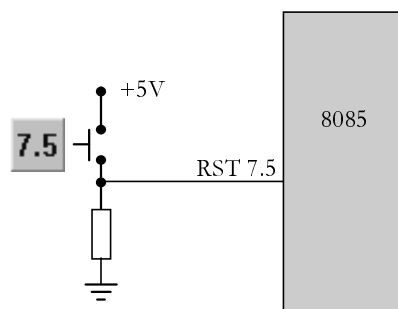


Fig. 15.1 – Circuito elétrico simulado da tecla RST 7.5

- quando atende uma dessas interrupções, o processador desvia o processamento do programa para o endereço correspondente, de acordo com a tabela 5.1, onde começam, a rigor, os tratadores. É preciso lembrar, porém, que estes endereços não correspondem à região de memória RAM do Abacus e que por isso este se comporta como se lá existissem desvios para endereços de RAM, de acordo com o exposto na seção 10.4.3;
- embora seja possível colocar os tratadores nos endereços definidos no Abacus, isto nem sempre é interessante, pois em geral esses endereços são escolhidos de forma a ficarem próximos uns dos outros, deixando pouco lugar para os tratadores. O que se faz então é escrever o tratador após o código do programa principal e identificar seu ponto de entrada com um label, por exemplo HNDLR, do inglês *handler* (tratador). Na posição

correspondente ao desvio programado no Abacus coloca-se então uma instrução JMP HNDLR, que desvia o processamento para o lugar correto.

15.2 Tratadores de interrupção na prática

15.2.1 Contador decimal com inibidor por interrupção

O programa a seguir demonstra a ocorrência do desvio para o tratador quando do botão da RST 7.5 é acionado. Neste exemplo, não existe um tratador de interrupção propriamente dito, mas apenas uma instrução HLT, que pára o processador imediatamente após o toque do botão RST 7.5.

Note que o código chama a sub-rotina DELAY, que faz parte das sub-rotinas simuladas em ROM. Sua função é introduzir um atraso no processamento do programa, necessário para que se possa visualizar o que acontece no display. O valor do atraso introduzido pelo Abacus na execução do programa, em milissegundos, é igual a 100 vezes o valor colocado no registrador D antes da chamada de DELAY.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Ints85a.asm - Interrupcoes do 8085
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
MOSTRAA EQU 036EH
DELAY EQU 05F1H
RST7.5 EQU 20CEH
ORG 2000H
LXI SP,20C0H ; Inicializa pilha
MVI A,18H ; Habilita 5.5, 6.5 e 7.5
SIM
MVI A,00H
EI
LOOP: ADI 01H ; INR A nao serve, nao atualiza CY
DAA ; utiliza CY flag na correcao
PUSH PSW ; Salva contagem
CALL MOSTRAA
MVI D,05H ; Atraso de 500ms
CALL DELAY
POP PSW ; Recupera contagem
JMP LOOP
ORG RST7.5 ; Desvio da RST 7.5
HLT
RET
END

```

15.2.2 Como não fazer

A título de exercício, você deve modificar o programa anterior de modo que a contagem reinicie do valor 1 a cada ocorrência da interrupção RST 7.5. O propósito desta seção é o de apresentar alguns erros comuns cometidos na solução deste problema e alertar para os perigos que representam.

A primeira tentativa

Em geral, a primeira reação para escrever o tratador é simplesmente zerar o acumulador. A listagem ficaria assim:

```
ORG RST7.5      ; Desvio da RST 7.5
MVI A, 01H
RET
```

Esta abordagem, porém, contraria várias das regras básicas para escrever tratadores de interrupção, apresentadas na seção 5.5. O tratador não preserva o valor do registrador A, que está sendo utilizado para tentar estabelecer uma comunicação com o programa principal. No presente caso, o tratador funcionaria se a interrupção acontecesse, por exemplo, durante a execução da instrução DAA. Esta seria concluída, em seguida o acumulador seria alterado e a contagem recomençaria de 1. Suponha, porém, que a interrupção acontecesse durante a execução da instrução PUSH PSW. Neste caso, o PUSH seria feito, o acumulador seria alterado e a contagem 01H de fato chegaria a aparecer no display. Mas, logo em seguida, POP PSW restauraria o valor que A continha antes da interrupção, e a contagem saltaria de volta para o valor antigo.

Se o programa for executado num hardware real, a interrupção pode acontecer durante a execução da sub-rotina MOSTRAA. Se a interrupção acontecer num ponto em que esta sub-rotina acaba de preparar, por exemplo, um valor em A para inicializar o hardware do display e este for alterado pela interrupção, então o display não funcionará como esperado.

Em geral, a alteração de qualquer registrador por um tratador de interrupção pode ter consequências catastróficas.

Além disso, o tratador apresentado tem um outro problema, que acontece com frequência quando se está aprendendo a escrever tratadores de interrupção: de acordo com as regras da seção 5.5, é preciso cuidar para que as interrupções estejam habilitadas quando forem necessárias. O 8085 desabilita automaticamente as interrupções assim que entra no tratador, e por isso o programa acima só aceitaria a primeira interrupção, uma vez que não há código que volte a habilitá-las. O final de um tratador é, por isso, tipicamente:

EI
RET

Esta finalização será adotada deste ponto em diante.

Habilitando e desabilitando

Uma sugestão que também aparece com certa frequência é a de desabilitar as interrupções em pontos críticos mediante a inserção de instruções DI (disable interrupts) e EI (enable interrupts) no corpo do programa principal, de modo que este só possa ser interrompido quando for conveniente. Esta solução é utilizada em alguns casos muito especiais, mas em geral é desaconselhada, pois sofre de sérias desvantagens:

- é difícil prever todas as situações em que as interrupções devem ser inibidas;
- mesmo em programas pequenos, pode haver vários pontos onde isso é necessário;
- o programa passa a ser incapaz de reagir às interrupções durante vários intervalos, quando é justamente o contrário que se deseja;
- a manutenção do programa fica extremamente difícil, pois cada alteração exige uma nova análise dos pontos de inibição das interrupções e é portanto fonte de perigosos erros.

Substituindo o registrador por uma variável global

Outra tentativa é substituir o acumulador por uma variável global para fazer a contagem, como na listagem a seguir. Note que agora o tratador foi mudado de lugar, de acordo com o exposto na seção 15.1.

```
ORG 2000H
LXI SP,20C0H      ; Inicializa pilha
MVI A,18H         ; Habilita 5.5, 6.5 e 7.5
SIM
MVI A,00H
STA COUNT
EI
LOOP: LDA COUNT
      ADI 01H      ; INR A não serve, não atualiza CY
      DAA         ; Utiliza CY flag na correção
      STA COUNT
      CALL MOSTRAA
      MVI D,05H    ; Atraso de 500ms
      CALL DELAY
      JMP LOOP
HNDLR: PUSH PSW
      MVI A,01H
      STA COUNT
```



```

POP PSW
EI                ; Volta a habilitar as interrupções
RET
ORG RST7.5        ; Desvio da RST 7.5
JMP HNDLR
COUNT DB 00H      ; Variável global, armazena contagem
END

```

Aqui não existe mais a comunicação via registrador entre o tratador de interrupção e o programa principal. Agora, se a interrupção acontecer dentro da sub-rotina MOSTRAA, tudo funciona bem: o valor do acumulador é guardado na pilha no início do tratador e restaurado no final, de modo que a sub-rotina interrompida não percebe qualquer alteração nesse registrador quando o seu processamento é retomado. Mas uma análise mais cuidadosa mostra que surge um problema se a interrupção acontecer durante a execução das instruções LDA COUNT, ADI 01H ou DAA: inicialmente, o tratador guarda o conteúdo de A na pilha, faz com que o valor da contagem volte a 1 e restaura o valor de A. Logo em seguida, porém, o programa principal é retomado com o valor de A preservado, e a instrução STA COUNT faz com que a contagem assuma o valor que teria se a interrupção não tivesse ocorrido.

O resultado é um programa de comportamento errático, que funciona quando o usuário pressiona o botão que aciona a interrupção enquanto o programa está fora desse trecho, mas que de vez em quando parece não funcionar. No caso de execução em um sistema real, o mau funcionamento pode ser atribuído, erroneamente, a um mau contato do botão. Este tipo de erro pode passar despercebido nos testes e ter consequências graves.

Sujando a pilha

Também pode aparecer a idéia de fazer o programa saltar do tratador de volta para o início:

```

LXI SP,20C0H      ; Inicializa pilha
MVI A,18H         ; Habilita 5.5, 6.5 e 7.5
SIM
START: MVI A,00H
STA COUNT
EI
LOOP:  LDA COUNT
ADI 01H           ; INR A não serve, não atualiza CY
DAA              ; utiliza CY flag na correção
STA COUNT
CALL MOSTRAA
MVI D,05H        ; Atraso de 500ms
CALL DELAY
JMP LOOP

```

```
HNDLR:  EI                ; Volta a habilitar as interrupções
        JMP START
        ORG RST7.5        ; Desvio da RST 7.5
        JMP HNDLR
COUNT  DB 00H            ; Variável global, armazena contagem
END
```

Esta abordagem é das piores, pois viola diretamente a regra que diz que o tratador não deve sujar a pilha. O que acontece aqui é que o endereço de retorno da interrupção fica armazenado na pilha, pois não se executa a instrução RET. A cada interrupção, a pilha recebe mais um word que nunca é retirado. Para contornar este problema, há quem sugira retirar o endereço de retorno com uma instrução POP, como segue:

```
HNDLR:  POP B              ; Par BC recebe endereço de retorno
        EI                ; Volta a habilitar as interrupções
        JMP START
        ORG RST7.5        ; Desvio da RST 7.5
        JMP HNDLR
COUNT  DB 00H            ; Variável global, armazena contagem
END
```

Mas esta solução retira apenas o endereço de retorno da pilha e, dependendo do ponto em que a interrupção ocorreu, a pilha pode conter mais words. Por exemplo, se a interrupção acontece dentro da sub-rotina MOSTRAA, então a pilha contém o endereço de retorno para esta sub-rotina, mais as palavras que ela possa ter colocado na pilha, e só depois disso vem o endereço de retorno da interrupção. Por isso, em geral o número de words que deveriam ser retirados da pilha depende do instante em que ocorre a interrupção, o que inviabiliza a solução proposta. Além disso, esta abordagem recai no primeiro erro, pois o par BC está sendo alterado pelo tratador, e haverá problemas se esses registradores forem necessários em algum ponto do programa (incluindo as sub-rotinas chamadas).

Finalmente, há quem sugira simplesmente forçar a pilha a voltar ao estado inicial, de modo que não possa ficar suja:

```
HNDLR:  LXI SP,20C0H      ; Limpa a pilha completamente
        EI                ; Volta a habilitar as interrupções
        JMP START
```

Esta abordagem pode ser aplicável em alguns casos, mas ainda assim é extremamente desaconselhável, porque:

- este método aborta qualquer sub-rotina que esteja em execução e, em programas mais complexos, isso pode ser catastrófico. Suponha, por exemplo, uma sub-rotina que liga um certo equipamento, envia alguns comandos a ele e precisa desligá-lo corretamente depois disso, sob pena de

danificá-lo. Se esta sub-rotina for abortada pela interrupção, o procedimento correto não será realizado;

- o programa não utiliza qualquer filosofia de organização de tratadores de interrupção, como a apresentada a seguir.

15.2.3 Como fazer

A listagem a seguir apresenta uma solução elegante para o problema, que obedece a todas as regras para escrever tratadores de interrupção.

```

;Ints85b.asm - Interrupcoes do 8085
;Prof. Roberto M. Ziller - 04.01.2000
;Ints85b.asm - Interrupcoes do 8085
;Prof. Roberto M. Ziller - 04.01.2000
;Ints85b.asm - Interrupcoes do 8085
;Prof. Roberto M. Ziller - 04.01.2000
MOSTRAA EQU 036EH
DELAY EQU 05F1H
RST7.5 EQU 20CEH
ORG 2000H
LXI SP,20C0H ; Inicializa pilha
MVI A,18H
SIM ; Habilita 5.5, 6.5 e 7.5
MVI A,00H
STA STATE ; Flag de reset da contagem
STA COUNT ; Inicializa contagem
EI
LOOP: LDA COUNT
ADI 01H ; INR A não serve, não atualiza CY
DAA ; Utiliza CY na correção
STA COUNT
CALL MOSTRAA
MVI D,05H ; Atraso de 500ms
CALL DELAY
LDA STATE
CPI 00H ; Testa critério de reinicialização
JZ LOOP ; Continua contando se STATE = 0
MVI A,00H
STA COUNT ; Caso contrário, zera a contagem
STA STATE ; Reinicializa flag
JMP LOOP
HNDLR: PUSH PSW
MVI A,01H
STA STATE
EI
POP PSW
RET
STATE DB 00H ; Flag global alterado pelo tratador
COUNT DB 00H ; Contagem

```

```
ORG RST7.5      ; Desvio da RST 7.5
JMP HNDLR
END
```

A elegância desta solução vem da separação clara das tarefas do programa principal e do tratador de interrupção. Foi definida uma variável global, STATE, que funciona como mecanismo de comunicação entre ambos. O tratador se limita a fazer STATE = 1 e o programa principal testa essa variável quando lhe for conveniente.

IMPORTANTE: o tratador não tenta modificar o valor da contagem!

Cada vez que chega ao final do loop, o programa principal consulta o valor da variável STATE e decide se deve voltar diretamente ao início e mostrar o próximo valor da contagem (STATE = 0) ou reinicializar a contagem antes de voltar (STATE = 1). Neste caso, zera também a variável STATE, para que o pedido de reinicialização não fique pendente.

Com isso, a contagem não pode ser modificada de forma inesperada, pois é o próprio programa principal que faz a modificação. A tarefa do tratador é simplesmente a de sinalizar ao programa principal que o usuário acionou a interrupção. O programa principal é que decide quando e onde deve processar esta informação. O princípio aplicado aqui deve ser utilizado sempre que possível para escrever tratadores de interrupção, pois o isolamento entre tratador e programa principal leva naturalmente a um programa de boa qualidade.

15.3 Exercícios

1. Modifique o programa da seção 15.2.3 de modo que, alternadamente e a cada toque do botão RST 7.5, a contagem pare e continue de onde parou.
 - ◊ Dica: modifique o tratador de interrupção de modo que este faça de STATE um contador módulo 2, que avança a cada toque do botão RST 7.5. O programa principal só precisa testar o valor de STATE (que será, alternadamente, 0 ou 1) e decidir se fica preso em um loop de espera (STATE = 1) ou se continua a contagem normalmente (STATE = 0).
2. Implemente um contador de duas velocidades que alterne entre ambas ao toque do botão RST 7.5.
 - ◊ Dica: utilize o tratador de interrupção do exercício acima, sem modificações. O programa principal deve decidir, com base no valor de STATE, se o valor carregado no par DE para o atraso na

contagem deve ser grande ($STATE = 0$, velocidade baixa) ou pequeno ($STATE = 1$, velocidade alta).

3. Existe uma generalização simples do tratador definido no exercício 1, que consiste em transformar a variável $STATE$ num contador módulo n . Com isso, é possível implementar programas que alternem ciclicamente entre n comportamentos diferentes.

Escreva um programa que implemente um contador que alterne ciclicamente entre os seguintes modos de contagem decimal, todas módulo 100:

- ◊ ascendente mostrada no campo de dados;
 - ◊ descendente mostrada no campo de dados;
 - ◊ ascendente mostrada no campo de endereços;
 - ◊ descendente mostrada no campo de endereços.
4. Implemente um programa com dois contadores hexadecimais de dois dígitos que evoluem juntos. O primeiro deve aparecer no campo de endereços e iniciar a contagem no sentido ascendente; o segundo deve aparecer no campo de dados e contar no sentido descendente. A toque do botão RST 7.5, os sentidos de contagem dos dois contadores deverão se inverter.

16 Portas de entrada e saída

16.1 As portas de E/S do Abacus

O 8155 é um dispositivo periférico projetado para operar em conjunto com o 8085. Oferece três portas paralelas de entrada e saída, denominadas A, B e C, dois temporizadores e 256 bytes de memória RAM, conforme a figura 16.1.

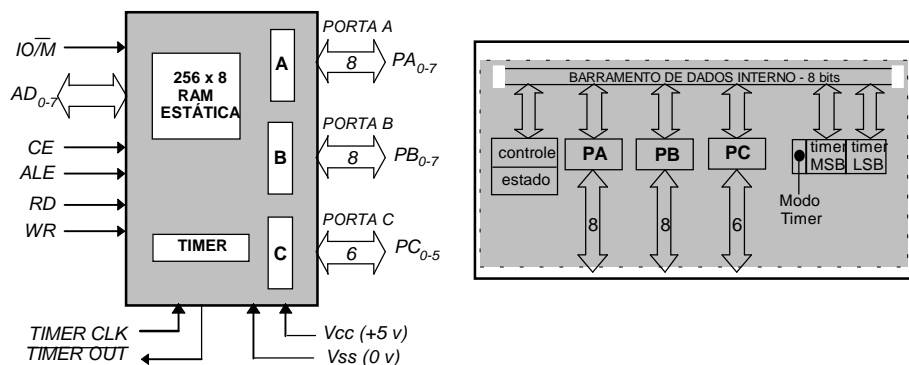


Fig. 16.1 – Representação do componente 8155

O Abacus simula a existência das portas A e B, cada uma de 8 bits, assim como do registrador de comando, que permite configurar essas portas como sendo de entrada ou de saída. À porta A está ligado o conjunto de chaves descrito na seção 10.3.6 e à porta B, o conjunto de leds descrito na seção 10.3.5. As ligações são mostradas na figura 16.2.

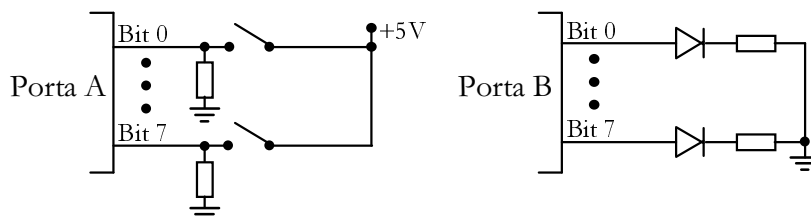


Fig. 16.2 – Ligação das chaves e dos leds ao 8155

As portas e o registrador de comando têm endereços fixos, dados pela tabela 16.1.

Registrador	End. 8155
COMMAND	20H
PORT A	21H (CHAVES)
PORT B	22H (LEDs)

Tab. 16.1 – Endereços das portas no Abacus

O objetivo do presente capítulo é demonstrar a utilização das portas A e B para fazer a leitura das chaves e o acender os leds.

16.2 A ligação do 8155 ao 8085

É bastante instrutivo examinar a ligação do 8155 ao microprocessador. Como mostra a figura 16.1, alguns pinos do 8155 têm nomes que também são encontrados no 8085: ALE, RD, WR, AD0-AD7 e IO/M. Além destes, são importantes CE (chip enable) e as portas de E/S (PA0 - 7, PB0 - 7 e PC0 - 5).

A coincidência de nomes de pinos entre os dois chips é proposital, e a ligação entre eles de fato se faz pela interconexão dos pinos de mesmo nome. Desta forma, o barramento que multiplexa dados e de endereços do 8085 é ligado diretamente ao 8155 através das linhas AD0-7. O 8155 é capaz de fazer a separação entre dados e endereços, pois recebe também o sinal ALE.

Além dos dados e endereços, o 8155 recebe os sinais RD, WR e IO/M, que permitem identificar operações de leitura e escrita em memória (no caso, os 256 bytes de RAM) ou nos periféricos (as portas e o temporizador). O sinal CE, finalmente, é ativado pela saída de um decodificador de endereços, de modo que o 8155 responde somente em uma determinada faixa de endereços, para evitar conflitos com outros periféricos.

Note ainda que os endereços da tabela 16.1 não são determinados pelo 8155, mas sim pelo projetista do hardware onde o componente é utilizado.

16.3 Configuração do 8155

Uma vez que o 8155 pode ser operado de diversas formas, é necessário configurá-lo de acordo com os objetivos que se tem em mente. Para tanto, é necessário escrever uma palavra de configuração num registrador deste

componente, chamado *registrador de comando* (*command register*). Este registrador tem oito bits; a função de cada bit está descrita na figura 16.3.

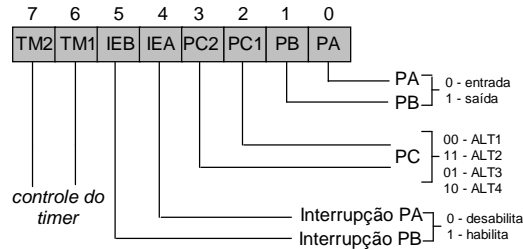


Fig. 16.3 – O registrador de comando do 8155

Os bits 0 e 1 configuram as portas A e B como sendo de entrada ou de saída e, no caso do Abacus, é interessante configurar a porta A como entrada e a porta B como saída, por causa das chaves e dos leds. A porta C não é simulada no Abacus, e por isso seu funcionamento não será detalhado aqui.

16.4 Exemplo e exercícios

1. O programa abaixo acende seqüencialmente os leds conectados à porta B do 8155. Note o uso da sub-rotina DELAY, que faz com que cada led permaneça aceso durante alguns instantes, para que se tenha o efeito visual desejado.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Leds.asm - Utilizacao de perifericos
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
COMMAND EQU 20H
LEDS EQU 22H
DELAY EQU 05F1H

ORG 2000H

LXI SP,20C0H ; Inicializa pilha
MVI A,02H ; Porta B como saida
OUT COMMAND
MVI A,01H

LOOP: OUT LEDS ; Acende o LED correspondente a A
RLC ; Rotaciona LED a acender
PUSH PSW
MVI D,02H ; Atraso de 200ms
CALL DELAY ; Espera para ver LED aceso
POP PSW
JMP LOOP
RET
END

```

2. Complete o programa do item 1 de forma que o valor colocado na porta de saída apareça também no campo de dados do mostrador.
3. Escreva um programa semelhante, mas que faça rotacionar um led apagado.
4. Inverta o sentido de rotação dos leds.
5. Implemente um contador cujo valor apareça em forma binária nos leds.
6. Escreva um programa que leia um número n do teclado, $0 \leq n \leq 8$, e que acenda, então, os n leds menos significativos (por exemplo, se $n = 5$, devem acender os leds correspondentes aos bits 4, 3, 2, 1 e 0). O programa deve então voltar ao início, aceitando novo valor para n .

◊ Dica: note que o número que deve ser colocado na porta de saída para acender os leds corresponde a $2^n - 1$. Este valor pode ser calculado em um loop, utilizando as instruções RLC e ORI 01H. Uma outra solução possível é utilizar STC e RAL.

7. O programa abaixo controla os leds de acordo com as chaves ligadas à porta A do 8155A.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Chaves.asm - Utilizacao de perifericos
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
COMMAND EQU 20H
CHAVES EQU 21H
LEDS EQU 22H
ORG 2000H
MVI A,02H ; Porta A como entrada, B como saída
OUT COMMAND
LOOP: IN CHAVES ; Le chaves para o acumulador
OUT LEDS ; Acende os LEDs correspondentes
JMP LOOP
RET
END

```

Escreva um programa que controle os leds de 0 a 6 a partir das chaves de mesmo número. A chave 7 deve ser uma chave de controle, que determina como os demais leds devem reagir às chaves: com a chave 7 aberta, os leds e as chaves de 0 a 6 devem se comportar de forma que a uma chave fechada corresponda um led aceso e a uma chave aberta um led apagado; com a chave 7 fechada, esta relação deve se inverter. O led 7 deve ficar apagado quando a chave 7 estiver aberta e acender quando esta for fechada.

8. Escreva um programa que leia as três chaves menos significativas através da porta A e associe a cada combinação das chaves um número binário n , de 000 (chaves abertas) a 111 (chaves fechadas). A qualquer instante, devem ficar acesos sempre os n leds menos significativos.

17 Assembly para o PC

Este capítulo apresenta as ferramentas de desenvolvimento (assembler, linker, debugger) para programas em assembly escritos para PC's com sistema operacional Windows ou MS-DOS.

17.1 O assembler

O assembler utilizado para os exemplos é o MASM. A partir do programa-fonte, que deve estar em um arquivo-texto de extensão .asm, são gerados o arquivo com os códigos (.obj) e a listagem (.lst).

As principais diretivas utilizadas para escrever programas em ASM-86 são apresentadas na tabela 17.1.

NOME	FUNÇÃO
ASSUME	Associa registrador de segmento a um nome de segmento
END	Indica final da listagem e ponto de entrada
EQU	Define um sinônimo
ORG	Estabelece o endereço da próxima instrução ou var. global
SEGMENT	Indica início de segmento
ENDS	Indica final de segmento
DB	Reserva e inicializa área de um ou mais bytes
DW	Idem, um ou mais words
PROC	Indica início de sub-rotina
ENDP	Indica final de sub-rotina
PUBLIC	Torna um símbolo visível a partir de outros módulos
EXTRN	Informa o MASM de que um símbolo é definido fora do módulo que está sendo montado

Tab. 17.1 – Principais diretivas do ASM-86

O MASM pode ser chamado sem parâmetros, caso em que passa a solicitar as informações de que necessita. Outra forma, mais rápida, de chamá-lo é:

MAASM <nome do arquivo-fonte>,,,

Esta forma gera os arquivos .obj e .lst, além de um arquivo de referências cruzadas (.crf), que não é de interesse prático para programas pequenos.

17.2 O linker

O linker (LINK) recebe como entrada um ou mais arquivos-objeto¹ (.obj) e gera os seguintes arquivos:

- programa executável (.exe);
- mapa de linkagem (.map);
- tabela de símbolos (.sym).

Assim como o MASM, o LINK pode ser chamado sem parâmetros, para que solicite as informações de que precisa. A sintaxe alternativa é:

LINK <nome do arquivo-objeto principal> [<nome de outro arquivo-objeto>, ...],,,

Esta gera os arquivos .map e .exe.

17.3 O debugger

O Symdeb é uma ferramenta utilizada na análise e na depuração de programas executáveis. O domínio das suas funções é de grande importância para o bom aproveitamento dos capítulos seguintes. Por isso, vale a pena investir um pouco de tempo no estudo dos seus principais comandos.

A sintaxe para a chamada do Symdeb é:

SYMDEB /opções [<drive>:]<nome>.<extensão>

São aceitos arquivos com extensão com, bin, exe e hex.

As opções são:

- K – habilita a tecla scroll lock como break;
- S – habilita chaveamento entre a tela de saída dos dados do programa e do Symdeb;
- I – imita o IBM-PC.

Destas, a opção S é a mais utilizada.

¹ O caso de programas com múltiplos arquivos é tratado no Anexo 7.

O sistema de numeração adotado por default em todos os comandos do Symdeb é o sistema hexadecimal. Números binários são aceitos como tais se seguidos do sufixo B (10101B) e números decimais devem ser seguidos do sufixo T (123T).

17.3.1 Comandos

Os comandos mais utilizados, que convém memorizar, são: R, DB, DW, DD, U, A, T, P e Q. Os exemplos abaixo ilustram a sua aplicação:

- R – mostra os registradores e os flags do processador;
- DB DS:0 L 20 – mostra os 32 (20H) primeiros bytes do segmento de dados endereçado por DS;
- DW DS:0 L 10 – mostra os mesmos 32 bytes dispostos na forma de words;
- DD DS:0 L 8 – mostra os mesmos 32 bytes dispostos na forma de double words;
- DW SS:SP L 8 – mostra os 8 últimos words colocados na pilha;
- U CS:IP L 15 – mostra ao todo 21 (15H) instruções, a partir da próxima instrução a ser executada;
- A [<addr>] – entra em modo assembler, em que é possível escrever pequenos programas;
- T – execução passo a passo;
- P – execução de uma sub-rotina ou tratador de interrupção num único passo;
- Q – Quit (encerra o programa).

O anexo 3 apresenta uma relação completa dos comandos do Symdeb, que também pode ser visualizada através do próprio programa, com o comando '?'.

17.4 Exercícios

1. Utilize o comando A (assemble) do Symdeb para criar o programa abaixo:

```
MOV AX,00FFH
MOV BX,AX
INC AL
INC BX
```

Em seguida, execute o programa passo a passo com o comando T (trace), observando o que acontece nos registradores. Este procedimento é útil para tirar pequenas dúvidas que possam surgir durante os estudos. Se precisar

saber o que acontece numa certa situação, imagine um pequeno exemplo e execute-o. É uma das melhores formas de estudar e aprender.

2. Monte o programa abaixo e carregue-o no Symdeb.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Mensagem.asm - Escrevendo uma mensagem na tela
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
PILHA    SEGMENT STACK
         DB 128 DUP(?)
PILHA    ENDS

DADOS    SEGMENT
MSG1     DB 'AGORA SEI ESCREVER MENSAGENS NA TELA DO
COMPUTADOR: $'
MSG2     DB 'MICROPROCESSADORES'
DADOS    ENDS

CODIGO    SEGMENT
ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
INICIO:  MOV AX,DADOS
         MOV DS,AX                ; INICIALIZACAO DE DS
         MOV AH,09H              ; SERVICO DO DOS
         MOV DX,OFFSET MSG1      ; APONTA PARA O TEXTO
         INT 21H                 ; EXECUTA FUNCAO DO DOS P/ MSG1
         MOV DX,OFFSET MSG2
         INT 21H                 ; IDEM, MSG2
         MOV AH,4CH              ; TERMINA E RETORNA AO DOS
         INT 21H                 ; SERVICO DO DOS
CODIGO    ENDS
         END INICIO

```

Execute-o passo a passo, acompanhando a evolução dos registradores e a saída na tela (comando \). As linhas

```

MOV AH,09H      ; FUNÇÃO DO DOS
LEA DX,MSG1     ; APONTA PARA O TEXTO
INT 21H         ; SERVIÇO DO DOS

```

chamam o tratador da interrupção 21H, responsável por um grande número de serviços oferecidos pelo sistema operacional DOS.

O tratador da interrupção 21H busca inicialmente em AH o número da função desejada. Com isso, este tratador pode oferecer até 256 funções diferentes. Algumas dessas funções têm subfunções, que o usuário especifica através de outro registrador, o que aumenta o limite do número de serviços possíveis para 65536. O número de serviços existentes é de algumas centenas. O anexo 4 descreve alguns desses serviços e mostra onde conseguir mais informações.

A função 09, utilizada no exemplo (porque AH=09H), serve para escrever uma string na tela do computador. Para tanto, é preciso obedecer à seguinte convenção de chamada:

- ◊ AH deve conter o valor 09H;
- ◊ o par DS:DX deve apontar para o primeiro byte da mensagem a ser escrita;
- ◊ a mensagem deve terminar por um caracter '\$', que não é escrito.

3. Identifique e corrija o erro no programa anterior.
4. Escreva um programa que leia caracteres do teclado e que escreva esses caracteres na tela.
 - ◊ Dica: consulte anexo 4 para ver o que fazem as funções 01H, 02H e 08H da interrupção 21H.
5. Modifique o programa anterior de modo que todas as letras apareçam como maiúsculas, independentemente de como foram digitadas.
 - ◊ Dica: lembre-se dos conceitos básicos sobre manipulação de bits.
6. Melhore o programa do item anterior para que seja possível ler strings e escrevê-las na tela. Encerre o programa se o caracter digitado for 'Enter'.
 - ◊ Dica 1: para descobrir o código gerado quando se digita 'Enter', faça um pequeno programa no Symdeb que leia uma tecla e execute-o passo a passo. Acompanhe o registrador que recebe o caracter digitado para descobrir o valor recebido. O mesmo procedimento pode ser usado para levantar o valor associado a qualquer tecla.
 - ◊ Dica 2: as teclas de funções especiais, como por exemplo F1, geram códigos de 2 bytes. Neste caso, o primeiro byte é sempre 00H. Assim, se v. precisar dessas teclas no seu programa, este deve ler o teclado e perguntar se o byte lido é nulo. Caso seja, deve imediatamente ler o teclado mais uma vez para ter o código completo.
7. Escreva um programa que utilize a função 09H da interrupção 10H, que oferece serviços do BIOS do PC. Esta interrupção também permite escrever caracteres na tela, mas além de especificar o caracter escrito, você também pode escolher sua cor. Consulte o anexo 4 para obter maiores detalhes sobre esta função.

18 Teste e depuração

18.1 Sub-rotinas near e far

O presente capítulo é um estudo dirigido para desenvolver as habilidades de teste e de busca de erros em programas.

O estudo explora as diferenças entre sub-rotinas near e far, que distinguem as chamadas de sub-rotinas feitas dentro de um mesmo segmento daquelas feitas de um segmento para outro.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;NearFar.asm - Um estudo sobre sub-rotinas near e far
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
PILHA    SEGMENT STACK
        DB 128 DUP(?)
PILHA    ENDS
DADOS    SEGMENT
MSG1     DB 'FOI CHAMADA UMA SUB-ROTINA NEAR$'
MSG2     DB 'FOI CHAMADA UMA SUB-ROTINA FAR$'
DADOS    ENDS
CODIGO    SEGMENT
        ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
        PUBLIC SHOW1, INICIO, SHOW2, ENTER
SHOW1    PROC NEAR          ; APRESENTA MSG EM DS:DX COM CRLF
        MOV AH,09H         ; WRITE STRING
        INT 21H
        CALL FAR PTR ENTER
        RET
SHOW1    ENDP
INICIO:   MOV AX,DADOS
        MOV DS,AX
        MOV DX,OFFSET MSG1
        CALL SHOW1
        MOV DX,OFFSET MSG2
        CALL FAR PTR SHOW2
        MOV AH,4CH         ; TERMINATE
        INT 21H
CODIGO    ENDS
```

```
ROTINAS SEGMENT
        ASSUME CS:ROTINAS
SHOW2   PROC FAR           ; APRESENTA MSG EM DS:DX COM CRLF
        MOV AH,09H        ; WRITE STRING
        INT 21H
        CALL FAR PTR ENTER
        RET
SHOW2   ENDP
ENTER   PROC FAR
        MOV AH,02H
        MOV DL,0DH        ; CARRIAGE RETURN (CR)
        INT 21H
        MOV AH,02H
        MOV DL,0AH        ; LINE FEED (LF)
        INT 21H
        RET
ENTER   ENDP
ROTINAS ENDS
        END INICIO
```

18.1.1 Explorando as listagens

Crie um arquivo com o programa-fonte dado acima. Utilizando a listagem do programa e os recursos oferecidos pelos programas MASM, LINK e Symdeb, responda as perguntas que seguem.

1. Quantos segmentos tem o programa e qual o tipo de cada um?
2. Qual o label que determina o ponto de entrada do programa e em que segmento ele fica?
3. Que listagem foi preciso consultar para responder a questão anterior?
4. Qual o offset do label onde o programa inicia? Responda esta questão utilizando pelo menos dois métodos diferentes e explique cada um deles.
5. Gere um arquivo de listagem (.lst) com o MASM na hora de montar o programa. Carregue o programa executável no Symdeb e faça um disassembly com o comando U. Compare os offsets das instruções mostradas com as geradas pelo MASM no arquivo de listagem.
6. Qual o valor inicial do ponteiro da pilha? Responda inicialmente baseado na listagem do programa-fonte e depois confirme sua resposta com o Symdeb. Descreva o procedimento adotado em cada um dos casos.
7. Gere um mapa de linkagem (.map) com o Link na hora de linkar o programa. Analise as informações contidas na listagem gerada. Por que esta não diz onde residirão os segmentos quando o programa for executado?

18.1.2 Explorando os recursos do Symdeb

Carregue o programa executável no Symdeb e siga os passos abaixo. Se você se perder e quiser recomençar, basta dar o comando L (load) para carregar novamente o arquivo e reinicializar os registradores.

1. Faça um disassembly do programa com o comando U; compare as instruções exibidas com a listagem do programa-fonte. Explique de onde vem o valor do símbolo “DADOS”, na linha onde inicia a execução do programa.
2. Note que o disassembly do item anterior não começou a ser feito no início do segmento (offset 0000H), mas sim do ponto de entrada do programa (000AH). Explique por que o ponto de entrada do programa não coincide com o início do segmento.
3. Faça um disassembly a partir do início do segmento de código com o comando U 0000 (ou simplesmente u0). Explique o significado das instruções encontradas.
4. Utilize o comando U <segmento>:<offset> para visualizar as instruções que compõem as sub-rotinas SHOW2 e ENTER. Explique como você consegue obter os valores de segmento e offset necessários.
5. Mostre os registradores e a primeira instrução a ser executada utilizando o comando R; preste atenção especial no par CS:IP, que aparece repetido no início da terceira linha, dando o endereço do ponto de entrada no programa.
6. Com base no disassembly do início do programa (comando U seguido do offset do ponto de entrada do programa), determine o valor que terá o segmento de dados. Utilize este valor no comando DB <segmento>:<offset> para visualizar as mensagens MSG1 e MSG2 e explique como determinar os valores do offset para cada uma das mensagens.
7. O programa utiliza o registrador DS para apontar para o segmento de dados. No entanto, se você utilizar o comando DB DS:<offset> para fazer a visualização das mensagens antes de iniciar a execução do programa, isso não funciona. Explique por quê.
8. Com o comando T, execute as primeiras duas instruções do programa e repita o comando DB DS:<offset> do exercício anterior. Explique por que agora o comando funciona.
9. Tendo executado as duas primeiras instruções, execute o comando U IP para ver as instruções seguintes. Continue a execução passo a passo, até

- chegar à a instrução CALL 0000. Observe os valores dos registradores CS, IP e SP antes e depois da execução desta instrução. Note que esta é uma chamada de sub-rotina near e que o endereço de retorno é constituído somente do offset. Como o segmento não muda, não há necessidade de armazená-lo.
10. Observe o conteúdo da pilha com o comando DW SS:SP L8; explique o significado das palavras mostradas.
 11. Você está agora dentro da sub-rotina SHOW1, que fica no mesmo segmento que o corpo principal do programa. Utilize o comando U IP para visualizar as instruções que seguem e avance até concluir a execução da instrução INT 21H. Utilize o comando \ para ver a mensagem que foi colocada na tela do usuário.
 12. Continue a execução passo a passo da sub-rotina SHOW1 até executar a próxima instrução CALL. Observe novamente o que acontece com os registradores CS, IP e SP quando esta instrução é executada e explique a diferença em relação à sua observação no item 9.
 13. Observe o conteúdo da pilha com o comando DW SS:SP L8; explique o valor das três primeiras palavras mostradas.
 14. Utilize o comando U IP para ver as instruções seguintes. Continue a execução, parando antes da instrução INT 21H. Observe a saída do usuário com o comando \, prestando especial atenção à posição do cursor. Volte então à tela do Symdeb e execute a instrução INT 21H. Observe novamente a saída e veja o que aconteceu com o cursor. Explique.
 15. Repita o procedimento do item anterior para a próxima instrução INT 21H.
 16. A próxima instrução a ser executada é RETF (return far). Você pode confirmar isso com o comando U IP. Execute esta instrução, observando com atenção o comportamento dos registradores CS, IP e SP. Mostre o conteúdo da pilha e explique o que aconteceu.
 17. Neste ponto, você voltou à sub-rotina SHOW1 e está prestes a executar sua última instrução, que é RET (confirme isso com U IP). Execute-a e observe novamente com atenção o que acontece com os registradores CS, IP e SP.
 18. Diga onde você está agora. Utilize U IP para se localizar, se necessário.
 19. Continue a execução do programa principal até a chamada da sub-rotina SHOW2. Tente prever o que acontecerá com os registradores CS, IP e SP quando a instrução CALL for executada, e depois confirme suas previsões.
 20. Tendo entrado na sub-rotina SHOW2, execute o comando U IP para ver seu código. Note que ela chama a sub-rotina ENTER, e que a chamada utilizada é do tipo FAR, embora ambas estejam no mesmo segmento de

código. Explique por que esta chamada não pode ser do tipo NEAR (pense no tipo de RET que existe no final da sub-rotina ENTER).

21. Continue a execução do programa passo a passo, até o final, observando sempre a saída do usuário antes e depois de cada instrução INT 21H e o comportamento da pilha a cada CALL, RETF ou RET.

22. Saia do Symdeb com o comando Q.

Repita este exercício em outro dia. Procure introduzir variações e explorar outros comandos do Symdeb, como E (edit) para modificar as mensagens apresentadas ou BP, BL, BD, BE e BC para trabalhar com breakpoints em conjunto com o comando G (go). Para maiores esclarecimentos, utilize o comando ? ou o anexo 3.

18.1.3 Introduzindo erros

Carregue o programa-fonte num editor de texto e experimente fazer as seguintes modificações:

1. Transforme a sub-rotina SHOW1 numa PROC FAR, mas não modifique a linha que a chama. Com isso, você estará fazendo um CALL NEAR para uma sub-rotina far. Salve o programa com um novo nome, monte-o com o MASM e gere a listagem (.lst). Compare a nova listagem com a anterior e explique as diferenças encontradas na chamada da sub-rotina e no código da instrução RET.
2. Crie o arquivo executável correspondente ao programa criado no item anterior e carregue-o no Symdeb. Execute-o passo a passo para ver o comportamento da pilha na chamada da função SHOW1. Explique por que o programa se perde depois de executar esta sub-rotina.
3. Partindo do arquivo original, transforme agora a sub-rotina SHOW2 numa PROC NEAR, modificando também a linha em que ela é chamada (em vez de CALL FAR PTR SHOW2 use somente CALL SHOW2) e salve o arquivo com um novo nome. Tente montar o programa com o MASM e analise a mensagem de erro gerada. Faça a montagem com geração de listagem (.lst) e analise essa listagem no ponto em que foi gerado o erro.
4. Modifique o arquivo gerado no item anterior, voltando a chamar a sub-rotina SHOW2 com CALL FAR PTR SHOW2, mas deixe a subrotina declarada como NEAR. Você estará assim introduzindo um erro grave no programa, pois a rotina será construída com um RET do tipo near no final e chamada com CALL FAR. Isso significa que a instrução CALL colocará na pilha os valores de CS e IP, mas que no final apenas o valor de IP será

recuperado pela instrução RET. Crie o arquivo executável correspondente e execute-o. Observe a saída, que pode conter resultados imprevisíveis.

5. Se necessário, reinicialize o seu microcomputador depois da experiência anterior. Carregue o programa que causou o problema no Symdeb e execute-o passo a passo, observando sempre o comportamento da pilha, que agora ficará defeituosa. Certifique-se de compreender exatamente o que acontece a cada CALL ou RET e procure explicar como o programa se perde. Tente explicar a mensagem que aparece na tela quando você executa o programa.
6. Partindo novamente do programa original dado no início do capítulo, utilize o editor de texto para remover a diretiva ASSUME do segmento rotinas. Salve o programa com um novo nome, crie o executável e carregue-o no Symdeb. Observe os valores dos segmentos de código e rotinas. Explique o que aconteceu.

18.1.4 Passando parâmetros

Os exercícios desta seção visam o explorar o mecanismo de passagem de parâmetros pela pilha, apresentado no capítulo 7.

1. Partindo do programa original da seção 18.1, identifique o tipo de passagem de parâmetros utilizado nas sub-rotinas SHOW1 e SHOW2.
2. Modifique as sub-rotinas SHOW1 e SHOW2 de modo que a passagem de parâmetros seja feita pela pilha. Faça também as modificações necessárias no programa principal para chamar as novas sub-rotinas.
3. Execute o programa obtido no item anterior passo a passo no Symdeb, verificando o comportamento da pilha. Certifique-se de compreender tudo que acontece!
4. Complete o programa apresentado no final da seção 7.3, de modo que o número de caracteres das duas mensagens escritas na tela também seja informado ao usuário.
5. Execute o programa criado no item anterior passo a passo no Symdeb, verificando as diferenças com o programa anterior. Observe o funcionamento da variável local que conta os caracteres escritos na tela.

19 Endereçamento baseado e indexado

19.1 Tabelas e matrizes

Este capítulo explora a representação de estruturas matriciais na memória. Como esta é um arranjo linear, essas estruturas não podem ser representadas sob forma bidimensional. É necessário transformá-las em um conjunto de tabelas, cada uma das quais armazena os dados de uma linha ou coluna.

O objetivo deste capítulo é mostrar como se pode utilizar o modo baseado e indexado para facilitar o endereçamento dos elementos de uma matriz. Os exercícios da seção 19.2 são úteis para melhorar o entendimento do uso de labels e de constantes e também para esclarecer as diferenças que existem entre estruturas de dados baseadas em bytes e em words.

Considere o programa abaixo, que define duas matrizes, M1 e M2, cada uma em um segmento de dados próprio e inicializadas com valores diferentes. O programa efetua a soma das duas matrizes, sobrescrevendo a primeira delas com o resultado.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Matrix.asm - Matrizes com endereçamento baseado indexado
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
M      EQU 10
N      EQU 20
PILHA  SEGMENT STACK
        DW 80H DUP(?)
PILHA  ENDS
M1     SEGMENT
        DB M * N DUP (10H)
M1     ENDS
M2     SEGMENT
        DB M * N DUP (25H)
M2     ENDS
```

```
CODIGO  SEGMENT
        ASSUME CS:CODIGO, SS:PILHA, DS:M1, ES:M2
        PUBLIC ROW, COL
INICIO:  MOV AX,M1
        MOV DS,AX
        MOV AX,M2
        MOV ES,AX
        MOV BX,0
ROW:     MOV SI,0
COL:     MOV AL,ES:[BX+SI]
        ADD [BX+SI],AL
        INC SI
        CMP SI,N
        JNZ COL
        ADD BX,N
        CMP BX,M*N
        JNZ ROW
        MOV AH,4CH
        INT 21H
CODIGO  ENDS
        END INICIO
```

19.2 Exercícios

1. Monte o programa e execute-o no Symdeb. Acompanhe o funcionamento dos registradores BX e SI e monitore as matrizes.
2. Você perceberá logo que é demorado avançar neste programa no modo passo a passo, sobretudo se desejar ver o registrador BX ser atualizado cada vez que SI termina de percorrer uma linha. Para tornar a depuração mais eficiente, pode-se criar um breakpoint no endereço da instrução que atualiza BX e depois utilizar o comando G (go). A execução será rápida, parando apenas quando o processador encontrar o endereço que tem o breakpoint. Utilizando o comando G várias vezes seguidas, você pode observar a matriz sendo processada linha a linha. Veja os comandos relacionados aos breakpoints no anexo 3.
3. Modifique o programa de modo que ambas as matrizes fiquem no mesmo segmento e utilize a forma de endereçamento baseado indexado com constante adicional para referenciá-las. A constante adicional deve ser utilizada como uma base conhecida em tempo de montagem.

Atenção: aqui existem várias maneiras de se conseguir um programa que funcione, mas nem todas têm a mesma qualidade. Os trechos de listagem abaixo dão exemplos comentados do que pode ser feito.


```
DADOS    SEGMENT
          DB M*N DUP (?)
          DB M*N DUP (?)
DADOS    ENDS
          ...
          MOV AL, [200+BX+SI]
          ADD [BX+SI], AL
```

Este método funciona, mas é ruim porque a constante 200 aparece dentro da instrução MOV. Se M e N forem alterados de modo que M*N passe a ser diferente de 200, o programador terá que se lembrar de atualizar esse valor dentro da instrução. Como isso pode ser facilmente esquecido, o código é vulnerável e tem baixa qualidade.

Para evitar este problema, poder-se-ia pensar em substituir

```
MOV AL, [200+BX+SI]
```

pela instrução

```
MOV AL, [M*N+BX+SI].
```

Esta abordagem também funciona e é certamente melhor do que a anterior, porque agora o valor da constante é corrigido automaticamente.

Mas ainda há um problema que pode acontecer: se alguém definir uma nova variável dentro do segmento de dados, antes da primeira matriz ou antes da segunda, os offsets das matrizes deixam de ser, respectivamente, 0 e M*N e aí o programa fica errado de novo. Por isso, a melhor solução consiste em adotar labels para as matrizes, como ilustrado abaixo:

```
DADOS    SEGMENT
M1        DB M*N DUP (?)
M2        DB M*N DUP (?)
DADOS    ENDS
          ...
          MOV AL, [M2+BX+SI]
          ADD [M1+BX+SI], AL
```

Note que M1 foi utilizado na instrução ADD, embora seu valor neste exemplo seja igual a zero. A utilização da constante, no entanto, garante que o programa continue funcionando corretamente mesmo que a ordem das matrizes dentro do segmento de dados seja invertida ou que outras variáveis sejam definidas antes da primeira matriz. O exemplo ilustra a utilidade dos labels, que devem ser empregados sempre que for necessário fazer referência a um offset dentro de qualquer segmento.

4. Modifique o programa construído no item anterior de modo que os valores dos elementos das matrizes sejam de 16 bits (words em vez de bytes). Observe o programa com cuidado, pois existem várias modificações a fazer!
5. Acrescente ao programa uma terceira matriz para receber os resultados da soma, preservando as duas matrizes que são somadas.

20 Desviando interrupções

20.1 Programas residentes

O sistema operacional MS-DOS não oferece suporte à multiprogramação, de modo que, normalmente, o usuário só pode instalar na memória um programa de cada vez. Constituem exceção a essa regra os programas chamados residentes, ou TSR's, do inglês *terminate and stay resident*. Um exemplo de programa residente é o *doskey*, utilizado para “lembrar” os comandos digitados pelo usuário e facilitar sua repetição. Os vírus também são TSR's.

Para que se torne ativo, um programa residente precisa ser colocado na memória por um programa instalador. Este aloca espaço para o código que deve permanecer residente, estabelece as condições para que este seja executado e então retorna ao prompt do sistema operacional. Essas condições de execução podem variar bastante de um programa para outro, e é comum que incluam contagem de tempo ou detecção de combinações de teclas, às quais o programa reage. No caso do *doskey*, por exemplo, cada comando digitado pelo usuário e confirmado com a tecla Enter é armazenado em uma lista. As teclas de setas ↑ e ↓ passam a ter a função de percorrer essa lista, colocando os comandos já digitados novamente à disposição do usuário.

Os TSR's conseguem provocar esse tipo de mudança no comportamento do sistema através da alteração de um ou mais tratadores de interrupção. Para tanto, modificam a tabela de interrupções de modo a desviar o tratamento da interrupção em questão para um trecho de código residente, que age como um tratador alternativo. Utilitários simples, como calculadoras e agendas que aparecem ao toque de uma combinação de teclas, costumam modificar os tratadores do relógio ou do teclado, mas o mesmo princípio se aplica também a programas mais sofisticados, como compactadores de disco e gerenciadores de memória, que implementam algoritmos mais sofisticados.

20.2 DOS idle interrupt

Existem casos em que um programa residente deve ser ativado assim que não houver outra tarefa a ser executada, a fim de aproveitar o tempo ocioso do processador. Esta é a situação que se tem quando um programa aguarda que o usuário digite algum comando, ou quando não há qualquer programa carregado para execução. Este mecanismo pode ser útil, por exemplo, para implementar TSR's de diagnóstico, que podem verificar o hardware instalado enquanto não há outras tarefas pendentes.

Uma vez que, por hipótese, não há qualquer ação do usuário que possa ser detectada para dar início à execução do programa residente, é preciso que exista uma outra forma de ativá-lo quando o sistema operacional entrar em repouso. É esta a função da interrupção número 28H do DOS, conhecida por *DOS idle interrupt* (interrupção de repouso), cujo tratador é chamado continuamente sempre que não há outras tarefas para processar. Por default, esse tratador tem apenas uma instrução IRET, de maneira que nada acontece se não houver um programa TSR instalado que o modifique.

O programa da seção 20.4 modifica o tratador da interrupção 28H para ilustrar o funcionamento do mecanismo de interrupção do 8086.

20.3 DOS idle interrupt e o Windows

A existência da interrupção de repouso do DOS criou um problema para o Windows. Como este é um sistema que oferece suporte à multiprogramação, o fato de uma janela do MS-DOS estar em repouso não significa que não haja outras tarefas a serem executadas. Portanto, se para cada janela MS-DOS aberta, o Windows tivesse que chamar continuamente o tratador da interrupção 28H – que normalmente executa apenas a instrução de retorno –, o desempenho das outras aplicações poderia ser prejudicado sem necessidade.

Por causa disso, existe uma diferença entre a execução do MS-DOS a partir do Windows e o caso em que se inicializa o computador apenas no modo MS-DOS. No primeiro caso, as chamadas do tratador da interrupção de repouso cessam após alguns instantes de inatividade. Assim, o programa da seção 20.4 entra em repouso logo depois que cessa a atividade do teclado. Isto não impede que se observe seu funcionamento, mas pode ser interessante reinicializar o computador no modo MS-DOS para observar a diferença.

20.4 O programa

O programa a seguir desvia o tratador da interrupção 28H para um trecho de código próprio, que permanece residente e implementa um contador módulo 10. Sua contagem aparece na tela, na posição atual do cursor, permitindo assim acompanhar as ocorrências dessa interrupção.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Ints86.asm - Interrupcoes do 8085
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
PILHA    SEGMENT STACK
        DW 128 DUP (?)
PILHA    ENDS
DADOS    SEGMENT
CONT     DB 0          ; CONTAGEM
ATRASO   DB 0          ; ATRASO
DADOS    ENDS
CODIGO    SEGMENT
        ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
INICIO:  MOV AX,CS
        MOV DS,AX          ; DS = CS
        MOV AH,25H         ; FUNCAO SET INT VECTOR
        MOV AL,28H         ; VETOR A MODIFICAR
        MOV DX,OFFSET CONTA ; OFFSET DO TRATADOR
        INT 21H            ; SERVICO DO DOS
        MOV AL,3           ; RETURN CODE
        MOV DX,0FFH        ; PARAGRAPHS TO KEEP RESIDENT
        MOV AH,31H         ; TSR
        INT 21H            ; SERVICO DO DOS
CONTA:   PUSH AX
        PUSH BX
        PUSH CX
        PUSH DS
        MOV AX,DADOS
        MOV DS,AX          ; DS -> DADOS
        INC ATRASO
        CMP ATRASO,00H
        JNE SAIDA
        MOV AL,CONT
        INC AL
        CMP AL,10
        JNE OK
        XOR AL,AL          ; ZERA CONTAGEM
OK:      MOV CONT,AL
        OR AL,30H          ; TRANSFORMA EM ASCII
        MOV AH,09          ; FUNCAO "WRITE CHAR" DA INT 10H
        MOV BH,0           ; PAGINA DE VIDEO
        MOV BL,7           ; ATRIBUTO

```

```
MOV CX,0001H      ; NUMERO DE CARACTERES
INT 10H            ; SERVICO DO BIOS
SAIDA: POP DS
                POP CX
                POP BX
                POP AX
                IRET
CODIGO ENDS
                END INICIO
```

20.5 Exercícios

1. Com o auxílio do Symdeb, faça um levantamento do código do tratador da interrupção 28H do DOS, antes de carregar qualquer programa residente.
2. Execute o programa acima a partir do prompt do DOS. Verifique novamente o endereço do tratador da interrupção 28H e o código que se encontra lá. Explique o que aconteceu.
3. Reinicialize o computador e carregue o programa no Symdeb. Execute-o passo a passo até que o tratador seja instalado e a contagem apareça na posição do cursor. Sem sair do Symdeb, encontre a posição da instrução INT 10H e substitua esta instrução por duas instruções NOP. Explique o que aconteceu.

◊ Dica: não utilize os comandos A (assemble) ou E (edit) do Symdeb para fazer as alterações. Se fizer isso, você não conseguirá alterar os dois bytes que constituem a instrução INT 10H de uma só vez e, durante o intervalo em que o primeiro byte tiver sido substituído pelo opcode da instrução NOP (90H) e o segundo byte ainda não, a instrução decodificada pelo processador fará o programa se perder. Para fazer a substituição dos dois bytes ao mesmo tempo, utilize o comando F (fill). Por exemplo, se a instrução INT 10H estiver em 1534:0043H, o comando é:

```
F 1534:43,44 90
```

4. O que acontece se você voltar a colocar a instrução INT 10H na posição dos dois NOP's?
◊ Dica: agora o caminho mais fácil é utilizar o comando A (assemble).
5. Experimente variar os valores de BL e CX na chamada para a interrupção 10H para criar efeitos coloridos, de acordo com as informações do anexo 4.

21 Trabalhando com strings

21.1 Inicialização e cópia de tabelas

O objetivo desta seção é analisar o emprego das instruções STOSW e MOVSW do 8086 para inicializar uma área de memória e copiar uma tabela de um segmento para outro.

O programa a seguir reserva uma área de trabalho (BUFFER) e utiliza a instrução STOSW para inicializá-la. Em seguida, coloca uma série de valores na pilha e copia esses valores para a área de trabalho com a instrução MOVSW.

A fim de manter o programa simples, não foram incluídas entrada nem saída de dados. A análise do funcionamento deve ser feita com a ferramenta Symdeb. Um exemplo completo, com entrada, processamento e saída, pode ser encontrado na seção 21.3.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Movs.asm - Cópia de strings
;Prof. Roberto M. Ziller - 04.01.2000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
WCOUNT EQU 10H
PILHA    SEGMENT STACK
          DW 80H DUP (?)
PILHA    ENDS

DADOS    SEGMENT
BUFFER   DW WCOUNT DUP (?)
DADOS    ENDS

CODE     SEGMENT
          ASSUME CS:CODE,SS:PILHA,ES:DADOS
          PUBLIC SUJAR, START
START:   MOV AX,DADOS
          MOV ES,AX
          LEA DI,BUFFER
          MOV CX,WCOUNT
          MOV AX,1111H      ; VALOR INICIAL DOS WORDS DE BUFFER
          CLD               ; STRINGS MOVIDAS COM AUTO-INCREMENTO
REP      STOSW
```

```
MOV CX,WCOUNT
SUJAR:  PUSH CX          ; COLOCA WCOUNT WORDS NA PILHA
LOOPNE SUJAR
        MOV AX,SS
        MOV DS,AX        ; DS = SS
        MOV SI,SP        ; SI = TOP OF STACK
        MOV CX,WCOUNT
        LEA DI,BUFFER
REP     MOVSW             ; WCOUNT WORDS DO STACK P/ BUFFER

        ADD SP,2*WCOUNT ; LIMPA A PILHA
        MOV AH,4CH
        INT 21H          ; FIM
CODE    ENDS
        END START
```

21.2 Exercícios

1. Monte o programa acima e carregue-o no Symdeb.
2. Estude a listagem e identifique as seções do programa que fazem a inicialização da área de trabalho, a colocação de valores na pilha e sua cópia para a área de trabalho.
3. Coloque um breakpoint no lugar da instrução que inicializa a área de trabalho e verifique o conteúdo dessa área imediatamente antes da execução dessa instrução. Execute-a com o comando T e verifique novamente o conteúdo da área de trabalho.
4. Qual o critério de parada do loop que “suja” a pilha? Que valores você espera que sejam colocados lá? Continue a execução do programa passo a passo e verifique suas previsões.
5. Avance até a execução da instrução que copia o conteúdo da pilha para a área de trabalho. Tente prever o novo conteúdo dessa área após a execução da instrução MOVSW e depois verifique suas previsões.
6. Escreva um programa que copie a tabela de vetores de interrupção para uma área de trabalho e que verifique então se essa área contém um vetor com o valor do offset igual a 0000H. A busca deve ser feita utilizando a instrução SCASW e não deve ser encerrada se houver algum segmento igual a 0000H. O programa deve apresentar como resultado o número do primeiro vetor encontrado cujo offset é zero.

21.3 Varredura de strings

O programa a seguir solicita ao usuário que digite uma sequência de até 100 caracteres e, em seguida, um caracter de busca. A sequência de caracteres é então varrida em busca da primeira ocorrência desse caracter com a instrução SCASB. Assim que for encontrado um caracter igual ao caracter de busca, o programa apresenta uma mensagem dizendo em que posição (de 00 a 99) o caracter foi encontrado. Caso o caracter de busca não faça parte da sequência, o programa apresenta uma mensagem dizendo que o caracter não foi encontrado.

```

;Scas.asm - Varredura de strings
;Prof. Roberto M. Ziller - 04.01.2000
ENTER    EQU 0DH
PILHA    SEGMENT STACK
        DW 100 DUP (?)
PILHA    ENDS
DADOS    SEGMENT
MSGSTR   DB 'ENTRE COM A STRING: $'
MSGCHAR  DB 'ENTRE COM O CARACTER A PROCURAR: $'
MSGFND   DB 'O CARACTER FOI ENCONTRADO NA POSICAO $'
MSGNOT   DB 'O CARACTER NAO FOI ENCONTRADO.'
CRLF     DB 0DH, 0AH, '$'
BUFFER   DB 101 DUP (?)
DADOS    ENDS
CODE     SEGMENT
        ASSUME CS:CODE,ES:DADOS,SS:PILHA
START:   MOV AX,DADOS
        MOV DS,AX
        MOV ES,AX
        LEA AX,MSGSTR
        PUSH AX
        CALL SHOW          ;SOLICITA STRING
        ADD SP,2
        LEA AX,BUFFER
        PUSH AX
        CALL GETSTR        ;STRING EM BUFFER E COMPRIMENTO EM CX
        ADD SP,2
        LEA AX,MSGCHAR
        PUSH AX
        CALL SHOW          ; SOLICITA CHARACTER
        ADD SP,2
        CALL GETCHAR       ; CHARACTER EM AL
        LEA DI,BUFFER
        CLD

```

```

REPNE SCASB          ; VARRE STRING EM BUSCA DO CHARACTER
      JNZ NO
      LEA AX,MSGFND
      PUSH AX
      CALL SHOW      ; CHARACTER ENCONTRADO
      ADD SP,2
      SUB DI,OFFSET BUFFER
      DEC DI          ; DI = OFFSET DO CHARACTER EM BUFFER
      MOV AX,DI
      MOV BL,10
      DIV BL          ; DIVIDE OFFSET DO CHARACTER POR 10;
      MOV DX,AX        ; DEZENA EM DL E UNIDADE EM DH
      ADD DX,3030H     ; CONVERTE P/ ASCII
      MOV AH,02H
      INT 21H          ; APRESENTA DEZENA
      MOV DL,DH
      INT 21H          ; APRESENTA UNIDADE
      MOV DL,'.'
      INT 21H          ; PONTO FINAL DA FRASE
      JMP FIM

NO:    LEA AX,MSGNOT
      PUSH AX
      CALL SHOW      ; CHARACTER NAO ENCONTRADO
      ADD SP,2

FIM:   MOV AH,4CH
      INT 21H

SHOW   PROC NEAR
      PUSH BP
      MOV BP,SP
      MOV AH,09H
      LEA DX,CRLF
      INT 21H
      MOV DX,[BP+4]
      INT 21H
      POP BP
      RET

SHOW   ENDP

GETSTR PROC NEAR
      PUSH BP
      MOV BP,SP
      MOV BX,[BP+4]
      MOV CX,00H
      LOOP: MOV AH,01H
            INT 21H
            CMP AL,ENTER
            JZ OK
            MOV [BX],AL
            INC BX          ; OFFSET DO CHARACTER EM BUFFER
            INC CX          ; NUMERO DE CARACTERES LIDOS
            CMP CX,100

```

```
                JB LOOP
OK:             MOV BYTE PTR [BX], '$'
                POP BP
                RET
GETSTR          ENDP
GETCHAR         PROC NEAR
                MOV AH, 01H
                INT 21H
                RET
GETCHAR         ENDP
CODE            ENDS
                END START
```

21.4 Exercícios

1. Monte e execute o programa acima e verifique que ele funciona conforme descrito.
2. Estude a listagem e identifique as seguintes seções no programa: entrada de dados, processamento e saída.
3. Carregue o programa no Symdeb e coloque um breakpoint na posição em que a busca está prestes a se iniciar. Execute o programa até este ponto e utilize o comando de visualização do conteúdo da memória para observar a variável BUFFER. Observe também os valores dos registradores AL e CX, e certifique-se de compreender seu papel no programa.
4. Continue a execução do programa passo a passo. Observe como o registrador DI é preparado para apontar para o início da string, antes da execução da linha REPNE SCASB. Qual a importância do registrador CX na execução dessa linha?
5. Como deve ser interpretado o conteúdo do registrador DI imediatamente após a execução da linha REPNE SCASB? Relacione sua resposta com o valor do zero flag.
6. Explique de forma simples e objetiva como o programa faz para obter o número decimal de 00 a 99 que representa a posição do character na string. Comece sua explicação imediatamente após a chamada da sub-rotina SHOW para apresentar a mensagem MSGFND.
7. Modifique o programa acima para apresentar todas as ocorrências de um character numa string, e não apenas a primeira.
8. Modifique o programa de modo que seja possível utilizar a tecla backspace para corrigir a string que está sendo digitada.

Anexo 1 – Instruções do 8085

1.1 Abreviaturas utilizadas na tabela de instruções

Abreviatura	Significado
dado	Constante de 8 bits
end	Endereço de 16 bits
p	Par de registradores (BC, DE ou HL)
r	Registrador (A, B, C, D, E, H ou L)
Z	Zero flag
S	Sign flag
P	Parity flag
CY	Carry flag
A	Auxiliary Carry flag
-/*/0/1	Não afeta/afeta/zera/seta flag

1.2 Diretivas

Nome	Significado
DB <valor>(,...)	Define byte
END	End
EQU	Equate
ORG <endereço>	Origin

1.3 As instruções

Esta seção apresenta o conjunto de instruções do 8085. A coluna “Hexa” contém o código hexadecimal de cada instrução; a coluna “ZSPCA” diz como cada instrução afeta os flags, de acordo com a convenção dada acima.

A fim de limitar o tamanho da tabela, algumas instruções foram agrupadas numa mesma linha. É o caso, por exemplo, da instrução *CMP <registrador>*. Para associar os opcodes aos registradores corretos, é preciso saber que os opcodes que se referem aos registradores de A até L e ao endereçamento indireto da memória ficam sempre na seguinte ordem: B, C, D, E, H, L, M e A. Assim, no caso da instrução acima, o opcode B8H significa CMP B, e o opcode BFH significa CMP A. No caso dos pares de registradores a ordem é BC, DE, HL (e SP, se for o caso).

Mnem.	Operando	Opcode	ZSPCA	Descrição
ACI	dado	CE	*****	Soma dado imediato e CY a A
ADC	r	88-8F	*****	Soma registrador e CY a A
ADC	M	8E	*****	Soma byte endereçado por HL e CY a A
ADD	r	80-87	*****	Soma registrador a A
ADD	M	86	*****	Soma byte endereçado por HL a A
ADI	dado	C6	*****	Soma dado imediato a A
ANA	r	A0-A7	*****	A = AND de A com registrador
ANA	M	A6	*****	A = AND de A com byte endereçado por HL
ANI	dado	E6	*****	A = AND de A com dado imediato
CALL	end	CD	-----	Chamada de sub-rotina
CC	end	DC	-----	CALL se CY = 1
CM	end	FC	-----	CALL se S = 1
CMA		2F	-----	Complementa A
CMC		3F	-----	Complementa CY
CMP	r	B8-BF	*****	Compara A com registrador
CMP	M	BE	*****	Compara A com byte endereçado por HL
CNC	end	D4	-----	CALL se CY = 0
CNZ	end	C4	-----	CALL se Z = 0
CP	end	F4	-----	CALL se S = 0
CPE	end	EC	-----	CALL se P = 1 (parity even)
CPI	dado	FE	*****	Compara A com dado imediato
CPO	end	E4	-----	CALL se P = 0 (parity odd)
CZ	end	CC	-----	CALL se Z = 1
DAA		27	*****	Ajuste de A após soma BCD
DAD	B	09	---*-	Soma o par BC a HL
DAD	D	19	---*-	Soma o par DE a HL
DAD	H	29	---*-	Soma o par HL a HL
DAD	SP	39	---*-	Soma SP a HL
DCR	A	3D	***-*	Decrementa A
DCR	B	05	***-*	Decrementa B
DCR	C	0D	***-*	Decrementa C
DCR	D	15	***-*	Decrementa D
DCR	E	1D	***-*	Decrementa E
DCR	H	25	***-*	Decrementa H
DCR	L	2D	***-*	Decrementa L
DCR	M	35	***-*	Decrementa byte endereçado por HL
DCX	p	0B,1B,2B,3B	-----	Decrementa par de registradores ou SP
DI		F3	-----	Desabilita interrupções
EI		FB	-----	Habilita interrupções
HLT		76	-----	Pára o processamento
IN	end	DB	-----	Lê byte do end. de I/O fornecido p/ A
INR	A	3C	***-*	Incrementa A
INR	B	04	***-*	Incrementa B
INR	C	0C	***-*	Incrementa C
INR	D	14	***-*	Incrementa D
INR	E	1C	***-*	Incrementa E
INR	H	24	***-*	Incrementa H
INR	L	2C	***-*	Incrementa L
INR	M	34	***-*	Incrementa byte endereçado por HL
INX	p	03,13,23,33	-----	Incrementa par de registradores ou SP
JC	end	DA	-----	Desvia se CY = 1
JM	end	FA	-----	Desvia se S = 1
JMP	end	C3	-----	Desvio incondicional
JNC	end	D2	-----	Desvia se CY = 0
JNZ	end	C2	-----	Desvia se Z = 0
JP	end	F2	-----	Desvia se S = 0
JPE	end	EA	-----	Desvia se P = 1 (parity even)
JPO	end	E2	-----	Desvia se P = 0 (parity odd)
JZ	end	CA	-----	Desvia se Z = 1
LDA	end	3A	-----	Lê byte do endereço fornecido para A
LDAX	B	0A	-----	Lê byte endereçado pelo par BC para A
LDAX	D	1A	-----	Lê byte endereçado pelo par DE para A

Mnem.	Operando	Opcode	ZSPCA	Descrição
LHLD	end	2A	-----	Lê word do endereço fornecido para HL
LXI	p,dado	01,11,21,31	-----	Copia dado imediato para par de regs
MOV	A,r	78-7F	-----	Copia reg para A
MOV	B,r	40-47	-----	Copia reg para B
MOV	C,r	48-4F	-----	Copia reg para C
MOV	D,r	50-57	-----	Copia reg para D
MOV	E,r	58-5F	-----	Copia reg para E
MOV	H,r	60-67	-----	Copia reg para H
MOV	L,r	68-6F	-----	Copia reg para L
MOV	M,r	70-77	-----	Copia reg para o byte endereçado por HL
MOV	r,M	46,4E,...,7E	-----	Copia byte endereçado por HL para reg
MVI	r,dado	06,0E,...,3E	-----	Copia dado imediato para reg
MVI	M,dado	36	-----	Copia dado para o byte endereçado por HL
NOP		00	-----	Operação nula
ORA	r	B0-B7	***00	A = OU de A com registrador
ORA	M	B6	***00	A = OU de A com byte endereçado por HL
ORI	dado	F6	***00	A = OU de A com dado imediato
OUT	endereço	D3	-----	Escreve A no endereço de I/O fornecido
PCHL		E9	-----	Copia HL para PC (causa um desvio)
POP	p	C1-D1-E1	-----	Desempilha um word e coloca no par p
POP	PSW	F1	*****	Desempilha um word e coloca em PSW
PUSH	p	C5-D5-E5	-----	Empilha par p
PUSH	PSW	F5	-----	Empilha PSW
RAL		17	---*-	Rotaciona A + CY para a esquerda
RAR		1F	---*-	Rotaciona A + CY para a direita
RC		D8	-----	RET se CY = 1
RET		C9	-----	Desempilha word e coloca em PC
RIM		20	-----	Copia a máscara de interrupção para A
RLC		07	---*-	Rotaciona A à esquerda; CY = MSB de A
RM		F8	-----	RET se S = 1
RNC		D0	-----	RET se CY = 0
RNZ		C0	-----	RET se Z = 0
RP		F0	-----	RET se S = 0
RPE		E8	-----	RET se P = 1 (parity even)
RPO		E0	-----	RET se P = 0 (parity odd)
RRC		0F	---*-	Rotaciona A à direita; CY = LSB de A
RST 0		C7	-----	Interrupção de software 0; PC = 0000H
RST 1		CF	-----	Interrupção de software 1; PC = 0008H
RST 2		D7	-----	Interrupção de software 2; PC = 0010H
RST 3		DF	-----	Interrupção de software 3; PC = 0018H
RST 4		E7	-----	Interrupção de software 4; PC = 0020H
RST 5		EF	-----	Interrupção de software 5; PC = 0028H
RST 6		F7	-----	Interrupção de software 6; PC = 0030H
RST 7		FF	-----	Interrupção de software 7; PC = 0038H
RZ		C8	-----	RET se Z = 1
SBB	r	98-9F	*****	Subtrai r e CY de A
SBB	M	9E	*****	Subtrai byte endereçado por HL e CY de A
SBI	dado	DE	*****	Subtrai dado imediato e CY de A
SHLD	endereço	22	-----	Copia par HL para os bytes end e end+1
SIM		30	-----	Copia A para a máscara de interrupção
SPHL		F9	-----	Copia HL para SP (afeta topo da pilha)
STA	endereço	32	-----	Copia A para o endereço fornecido
STAX	B	02	-----	Copia A para o byte endereçado por BC
STAX	D	12	-----	Copia A para o byte endereçado por DE
STC		37	---1-	Seta CY
SUB	r	90-97	*****	Subtrai r de A
SUB	M	96	*****	Subtrai byte endereçado por HL de A
SUI	dado	D6	*****	Subtrai dado imediato de A
XCHG		EB	-----	Permuta pares HL e DE
XRA	r	A8-AF	***00	A = XOR de A com r
XRA	M	AE	***00	A = XOR de A com byte endereçado por HL
XRI	dado	EE	***00	A = XOR de A com dado imediato
XTHL		E3	-----	Permuta HL com word no topo da pilha

Anexo 2 – Instruções do 8086

2.1 Abreviaturas, flags e diretivas

Short name	Meaning
a	Address
c	Count
d	Destination
e	Expression or string
p	I/O port
r	Register
s	Source
sr	Segment register (CS,DS,SS,ES)
sy	Symbol
t	Type of symbol

Flag	Short	Meaning
OF	O	Overflow Flag (Bit 11)
DF	D	Direction Flag (Bit 10)
IF	I	Interrupt enable Flag (Bit 9)
TF	T	Trap Flag (Bit 8)
SF	S	Sign Flag (Bit 7)
ZF	Z	Zero Flag (Bit 6)
AF	A	Auxiliary carry Flag (Bit 4)
PF	P	Parity Flag (Bit 2)
CF	C	Carry Flag (Bit 0)
Affected?	-/*0/1/?	Unaffected/affected/reset/set/unknown

Name	Function
ALIGN	Align to word boundary
ASSUME sr:sy(...)	Assume segment register name(s)
ASSUME NOTHING	Remove all former assumptions
DB e(...)	Define Byte(s)
DBS e	Define Byte Storage
DD e(...)	Define Double Word(s)
DDS e	Define Double Word Storage
DW e(...)	Define Word(s)
DWS e	Define Word Storage
END <label>	End and entry point of program
EXT (sr:)sy(t)	External(s) (t=ABS/BYTE/DWORD/FAR/NEAR/WORD)
LABEL t	Label (t=BYTE/DWORD/FAR/NEAR/WORD)
PROC t	Procedure (t=FAR/NEAR, default NEAR)

2.2 Tipos

ABS	Absolute value of operand
BYTE	Byte type operation
DWORD	Double Word operation
FAR	IP and CS registers altered
HIGH	High-order 8 bits of 16-bit value
LENGTH	Number of basic units
LOW	Low-order 8 bit of 16-bit value
NEAR	Only IP register need be altered
OFFSET	Offset portion of an address
PTR	Create a variable or label
SEGMENTO	Segment of address
SHORT	One byte for a JMP operation
SIZE	Number of bytes defined by statement
THIS	Create a variable/label of specified type
TYPE	Number of bytes in the unit defined
WORD	Word operation

2.3 As Instruções

Mnemonic	ODITSZAPC	Description
AAA	?---??*?*	ASCII Adjust for Add in AX
AAD	?---**?*?	ASCII Adjust for Divide in AX
AAM	?---**?*?	ASCII Adjust for Multiply in AX
ÅS	?---??*?*	ASCII Adjust for Subtract in AX
ADC d,s	*---*****	Add with Carry
ADD d,s	*---*****	Add
AND d,s	*---**?*?	Logical AND
CALL a	-----	Call
CBW	-----	Convert Byte to Word in AX
CLC	-----0	Clear Carry
CLD	-0-----	Clear Direction (increment)
CLI	--0-----	Clear Interrupt
CMC	-----*	Complement Carry
CMP d,s	*---*****	Compare
CMPS	*---*****	Compare memory at SI and DI
CWD	-----	Convert Word to Double in AX,DX
DAA	?---*****	Decimal Adjust for Add in AX
DAS	?---*****	Decimal Adjust for Subtract in AX
DEC d	*---*****-	Decrement
DIV s	?---?????	Divide (unsigned) in AX(,DX)
ESC s	-----	Escape (to external device)
HLT	-----	Halt
IDIV s	?---?????	Divide (signed) in AX(,DX)
IMUL s	*---????*	Multiply (signed) in AX(,DX)
IN d,p	-----	Input
INC d	*---*****-	Increment
INT	--00----	Interrupt
INTO	---*-----	Interrupt on Overflow
IRET	*****	Interrupt Return
JB/JNAE a	-----	Jump on Below/Not Above or Equal
JBE/JNA a	-----	Jump on Below or Equal/Not Above
JCZX a	-----	Jump on CX Zero
JE/JZ a	-----	Jump on Equal/Zero
JL/JNGE a	-----	Jump on Less/Not Greater or Equal
JLE/JNG a	-----	Jump on Less or Equal/Not Greater

Mnemonic	ODITSZAPC	Description
JMP a	-----	Unconditional Jump
JNB/JAE a	-----	Jump on Not Below/Above or Equal
JNBE/JÁ a	-----	Jump on Not Below or Equal/Above
JNL/JGE a	-----	Jump on Not Less/Greater or Equal
JNE/JNZ a	-----	Jump on Not Equal/Not Zero
JNLE/JG a	-----	Jump on Not Less or Equal/Greater
JNO a	-----	Jump on Not Overflow
JNP/JPO a	-----	Jump on Not Parity/Parity Odd
JNS a	-----	Jump on Not Sign
JO a	-----	Jump on Overflow
JP/JPE a	-----	Jump on Parity/Parity Even
JS a	-----	Jump on Sign
LAHF	-----	Load AH with 8080 Flags
LDS r,s	-----	Load pointer to DS
LEA r,s	-----	Load EA to register
LES r,s	-----	Load pointer to ES
LOCK	-----	Bus Lock prefix
LODS	-----	Load memory at SI into AX
LOOP a	-----	Loop CX times
LOOPNZ/LOOPNE a	-----	Loop while Not Zero/Not Equal
LOOPZ/LOOPE a	-----	Loop while Zero/Equal
MOV d,s	-----	Move
MOVS	-----	Move memory at SI to DI
MUL s	*---****	Multiply (unsigned) in AX(,DX)
NEG d	*---*****	Negate
NOP	-----	No Operation (= XCHG AX,AX)
NOT d	-----	Logical NOT
OR d,s	*---**?*	Logical inclusive OR
OUT p,s	-----	Output
POP d	-----	Pop
POPF	*****	Pop Flags
PUSH s	-----	Push
PUSHF	-----	Push Flags
RCL d,c	*-----*	Rotate through Carry Left
RCR d,c	*-----*	Rotate through Carry Right
REP/REPNE/REPZ	-----	Repeat/Repeat Not Equal/Not Zero
REPE/REPZ	-----	Repeat Equal/Zero
RET (s)	-----	Return from call
ROL d,c	-----	Rotate Left
ROR d,c	*-----*	Rotate Right
SAHF	-----	Store AH into 8080 Flags
SAR d,c	*---**?*	Shift Arithmetic Right
SBB d,s	*---*****	Subtract with Borrow
SCAS	*---*****	Scan memory at DI compared to AX
SEG r	-----	Segment register
SHL/SAL d,c	*---**?*	Shift logical/Arithmetic Left
SHR d,c	*---**?*	Shift logical Right
STC	-----1	Set Carry flag
STD	-1-----	Set Direction flag (decrement)
STI	--1-----	Set Interrupt flag
STOS	-----	Store AX into memory at DI
SUB d,s	*---*****	Subtract
TEST d,s	*---**?*	AND function to flags
WAIT	-----	Wait
XCHG r(,d)	-----	Exchange
XLAT	-----	Translate byte to AL
XOR d,s	*---**?*	Logical Exclusive OR

Anexo 3 – Comandos do Symdeb

A <addr> – Assemble

Converte mnemônicos digitados em código hexadecimal na memória.

Exemplo: A 1000

BP <addr> – Breakpoint

Cria um breakpoint no endereço especificado. Cada breakpoint criado recebe um número de 0 a 9. Os breakpoints definidos podem ser visualizados com o comando BL. Pode haver no máximo 10 breakpoints definidos ao mesmo tempo.

Exemplos: BP 2000:0200, BP CS:IP

BL – Breakpoint List

Lista os breakpoints criados pelo comando BP, apresentando o número do breakpoint, seu endereço e se está habilitado (símbolo **e**, de enabled) ou não (**d**, disabled). Os breakpoints podem ser habilitados e desabilitados com os comandos BE e BD, respectivamente.

Exemplo: BL

BC <n> – Breakpoint Clear

Elimina permanentemente o breakpoint *n*.

Exemplos: BC 2, BC * (limpa todos)

BD <n> – Breakpoint Disable

Desabilita o breakpoint *n*.

Exemplos: BD 5, BD * (desabilita todos)

BE <n> – Breakpoint Enable

Habilita o breakpoint *n*.

Exemplos: BE 5, BE * (habilita todos)

C <addr1> L<length> <addr2> – Compare

Compara dois blocos de memória.

Exemplo: C 2000:0100 L200 3000:0100

DA <addr> – DA <addr1> <addr2> – Dump ASCII

Lista os caracteres ASCII das posições de memória especificadas.

Exemplos: DA 2000:0100, DA 2000:0100 2000:0300

DB <addr> – DB <addr1> <addr2> – Dump Byte

Lista os conteúdos das posições de memória especificadas byte a byte, em hexadecimal e ASCII.

Exemplos: DB 2000:0100, DB 2000:0100 2000:0300

DW <addr> – DW <addr1> <addr2> – Dump Word

Lista os conteúdos em forma de words.

Exemplos: DW 2000:0100, DW 2000:0100 2000:0300

DD <addr> – DD <addr1> <addr2> – Dump Double Word

Lista os conteúdos em forma de double words.

Exemplos: DD 2000:0100, DD 2000:0100 2000:0300

DS <addr> – DS <addr1> <addr2> – Dump short real

Lista conteúdos interpretados como variáveis short real (FORTRAN).

DL <addr> – DL <addr1> <addr2> – Dump long real

Lista conteúdos interpretados como variáveis long real (FORTRAN).

DT <addr> – DT <addr1> <addr2> – Dump ten byte

Lista conteúdos interpretados como variáveis ten byte (FORTRAN).

D <addr>, D <addr1> <addr2> – Dump

Repete o último comando Dump executado (DB, DW, DD, ...).

Exemplos: D 2000:0100, D 2000:0100 2000:0300

E <addr> [value...] – Edit

Substitui o conteúdo de um ou mais bytes no endereço especificado.

Exemplos: E 2000:0100, E DS:BX 43 30 28

EW <addr> [value...] – Edit word

Substitui o conteúdo de um ou mais words.

F <addr> L<length> <value> – Fill

Preenche uma faixa de memória com um byte especificado.

Exemplo: F 2000:0100 L0200 00

G, G <addr>, G <addr1> <addr2> – Go

Executa o programa que está na memória a partir de CS:IP, ou do endereço especificado, ou de addr1 até addr2.

Exemplos: G, G CS:1234, G 2000:0100 2000:0300

H <value1> <value2> – Hex arithmetic

Soma e subtrai dois valores hexadecimais.

Exemplo: H 50 30

I – Input Port

Recebe e mostra um byte de uma porta.

Exemplo: I <número da porta>

L, L <addr> <drive> – Load

Carrega o arquivo especificado pelo comando N (v. abaixo).

M <addr1> <addr2> <addr3> – Move

Move bloco de memória entre addr1 e addr2 para addr3.

Exemplo: M 2000:0100 2000:0300 4000:0100

N <name> – Name

Especifica um arquivo para ser carregado ou salvo pelos comandos L e W.

Exemplo: N Lab0101.exe.

P – Proceed

Executa sub-rotina ou tratador de interrupção.

Q – Quit

Encerra o programa.

R, R <reg>, R <reg> = <v> – Register

Apresenta os registradores ou atribui ao registrador especificado o valor *v*.

Exemplos: R, R AX, R IP = 0100

S <addr1> <addr2> <values> – Search

Procura uma sequência de bytes dentro de uma faixa.

Exemplo: S 0100 0200 41 40 30

T, T<*n*> – Trace

Executa passo a passo uma ou *n* instruções.

Exemplos: T, T5

U <addr>, U <addr> L<length> – Unassemble

Interpreta o conteúdo da memória como instruções assembler e gera os mnemônicos correspondentes.

Exemplos: U 1234:5678, U CS:IP L 10

W, W <addr> <drive> – Write

Salva o arquivo com o nome especificado pelo comando N.

? – Help

Ajuda. Descreve resumidamente cada um dos comandos.

\ – User output

Apresenta a tela de saída do programa. Disponível somente se o Symdeb for chamado com a opção /S.

Anexo 4 – Serviços do DOS e do BIOS

Este anexo apresenta convenções de chamada de alguns serviços do DOS e do BIOS. Informações mais completas sobre estes e outros serviços podem ser obtidas em livros como [BK91] ou ainda na Internet. Consulte o item Links Interessantes na homepage que dá suporte ao livro.

Cada serviço corresponde a uma interrupção do PC e oferece uma ou mais funções, identificadas por um número entre 00H e FFH. Algumas destas funções têm subfunções, também numeradas de 00H a FFH, de forma que, teoricamente, pode-se ter até $256 * 256 = 65536$ subfunções para uma única interrupção. O número de funções e subfunções implementadas não é tão grande, mas chega a várias centenas. Por isso, é preciso ter à mão algum tipo de referência para trabalhar.

A fim de utilizar esses serviços é preciso seguir à risca as convenções de chamada definidas pelo fabricante do software. Estas convenções são expressas em termos dos valores a serem colocados nos registradores do processador antes de invocar a função desejada e, se for o caso, dos valores retornados.

4.1 Funções do BIOS

4.1.1 Serviços de vídeo (INT 10H)

Função 00H – SET VIDEO MODE

- AH = 00H;
- AL = modo (03H para VGA 80x25).

Obs.: normalmente, esta função também limpa a tela do monitor. Nos modos padrão IBM, o conteúdo da tela pode ser preservado colocando-se o bit mais significativo de AL em 1.

Função 01H – set cursor size

- AH = 01H;
- CH:
 - ◊ bit 7 = 0;
 - ◊ bits 6 e 5 = 00 (normal), 01 (invisível);
 - ◊ bits 4 a 0 = linha mais alta do caracter onde deve aparecer o cursor;
- CL, bits 4 a 0 = linha mais baixa do caracter onde deve aparecer o cursor.

V. também função 03H.

Função 02H – set cursor position

- AH = 02H;
- BH = número da página de vídeo, de 0 a 3 para o modo de vídeo 03;
- DH = linha (00H é a linha superior);
- DL = coluna (00H é a coluna da esquerda).

V. também funções 00H e 03H.

Função 03H – get cursor position and size

- AH = 03H;
- BH = número da página de vídeo (0 a 3 para modo de vídeo 03, v. função 00).

Registradores no retorno:

- CH = linha inicial do caracter contendo cursor;
- CL = linha final do caracter contendo cursor;
- DH = linha da tela que contém o cursor (00H é a linha superior);
- DL = coluna do caracter que contém o cursor (00H = esquerda).

V. também funções 01H e 02H.

Função 05H – select active display page

- AH = 05H;
- AL = página que deve se tornar ativa.

Função 09H – write character and attribute at cursor position

- AH = 09H;
- AL = caracter a ser escrito na tela;
- BH = número da página de vídeo (0 a 3 para VGA);

- BL = atributo de vídeo (v. figura 4.1);
- CX = número de vezes que o caracter deve ser escrito.

7	6	5	4	3	2	1	0
Bl	b	b	b	f	f	f	f

Fig. A4.1 – Atributo de vídeo em modo texto

O atributo de um caracter

Cada caracter que aparece na tela é descrito por dois bytes. O primeiro contém seu código ASCII e o segundo, que determina sua aparência, é denominado *atributo*. O bit 7 deste byte (blinking) determina se o caracter aparece piscando; e os bits 4-6 e 0-3 determinam, respectivamente, a cor do fundo sobre o qual o caracter será escrito e a cor do próprio caracter, de acordo com a tabela A4.1.

Cor	Valor	Utilização
Preto	0H	Caracter e fundo
Azul	1H	Caracter e fundo
Verde	2H	Caracter e fundo
Ciano	3H	Caracter e fundo
Vermelho	4H	Caracter e fundo
Magenta	5H	Caracter e fundo
Marrom	6H	Caracter e fundo
Cinza claro	7H	Caracter e fundo
Cinza escuro	8H	Somente caracter
Azul claro	9H	Somente caracter
Verde claro	AH	Somente caracter
Ciano claro	BH	Somente caracter
Vermelho claro	CH	Somente caracter
Magenta claro	DH	Somente caracter
Amarelo	EH	Somente caracter
Branco	FH	Somente caracter

Tab. A4.1 – Valores dos campos do atributo

Função 0AH – write character only at cursor position

- AH = 0AH;
- AL = caracter a ser escrito na tela;
- BH = número da página de vídeo (0 a 3 para VGA);
- BL = atributo de vídeo;
- CX = número de vezes que o caracter deve ser escrito.

Função 0BH – set background / border color

- AH = 0BH;
- BH = 00H;
- BL = cor dor fundo / da borda;

4.2 Funções do DOS

4.2.1 Serviços gerais (INT 21H)

Função 01H – read character from standard input, with echo

- AH = 01H.

Registradores no retorno:

- AL = código ASCII do caracter lido.

V. também função 08H.

Função 02H – write character to standard output

- AH = 02H;
- DL = código ASCII do caracter a ser escrito.

Obs.: no monitor, o caracter aparece na posição atual do cursor.

V. também função 09H.

Função 08H – character input without echo

- AH = 08H.

Registradores no retorno:

- AL = código ASCII do caracter lido.

V. também função 01H.

Função 09H – write \$-terminated string to standard output

- AH = 09H;
- DS:DX = ponteiro far para o início da cadeia de caracteres a ser escrita.

Obs.: se a saída padrão for o monitor, a escrita inicia na posição atual do cursor. A escrita termina quando for encontrado o caracter '\$', que não é escrito.

V. também função 02H.

Função 25H – Set interrupt vector

- AH = 25H;

- AL = número do vetor de interrupção a ser modificado;
- DS:DX = novo valor para o vetor a ser modificado.

Função 31H – terminate and stay resident (TSR)

- AH = 31H;
- AL = return code;
- DX = número de parágrafos que devem permanecer residentes.

Função 4CH – terminate program

- AH = 4CH;
- AL = return code.

4.2.2 DOS Idle interrupt (INT 28H)

Esta interrupção é chamada pelo próprio sistema operacional sempre que este está esperando uma entrada do usuário. Foi criada com o propósito de permitir que programas TSR modifiquem seu tratador de interrupção (que, por default, consiste apenas de uma instrução IRET), a fim de que possam ser chamados enquanto se aguarda alguma ação do usuário.

Anexo 5 – Arquivos .com

No sistema operacional MS-DOS, o formato de arquivo executável mais comum, caracterizado pela extensão .exe, permite a utilização de toda a capacidade de endereçamento do processador. Por isso, pode ser utilizado para programas de qualquer tamanho suportado pelo sistema operacional.

Existe também um formato mais compacto, caracterizado pela extensão .com e que limita o tamanho total do programa (código, dados e pilha) a 64 kB. Neste formato, todos os registradores de segmento são inicializados pelo sistema operacional com o mesmo valor, de modo que o processador enxerga apenas um segmento. Código, dados e pilha devem ser dispostos de maneira a não interferirem uns com os outros, da mesma forma que no caso de processadores que endereçam apenas 64 kB de memória, como o 8085.

Programas com este formato devem obedecer às seguintes convenções:

- o arquivo-fonte não deve declarar segmento de pilha;
 - ◊ isto gera um warning de linkagem, que deve ser ignorado;
 - ◊ a diretiva ASSUME não deve fazer referência a SS;
 - ◊ mesmo assim, a pilha pode ser utilizada normalmente (SP é inicializado em FFFEh);
- a diretiva ASSUME deve referenciar CS e DS, ambos para o mesmo segmento;
 - ◊ não deve haver segmento separado de dados;
 - ◊ dados e código devem ficar dentro do mesmo segmento;
- nenhum registrador de segmento deve ser inicializado;
- arquivos .com têm um cabeçalho de 256 bytes (offsets 00 a FFh) no início do programa;
 - ◊ o segmento de código deve iniciar com ORG 100h;
- o endereço 100h tem que ser o ponto de entrada;
 - ◊ não pode haver dados ou sub-rotinas antes desse endereço.

A montagem e a linkagem do programa são feitas da mesma forma como no caso dos arquivos .exe, com os seguintes passos:

```
MASM <nome>, , , ,
```

```
LINK <nome>, , , ,
```

Depois disso, converte-se o arquivo .exe em um arquivo .com. O programa que faz esta conversão chama-se exe2bin.exe:

```
EXE2BIN <nome>.exe <nome>.com
```

O exemplo a seguir mostra um programa simples, escrito primeiramente como .exe e depois transformado em .com. Compare-os, observando as convenções apresentadas.

Versão .exe

```
PILHA    SEGMENT STACK
         DB 128 DUP(?)
PILHA    ENDS
DADOS    SEGMENT
MSG1     DB 'Este programa tem formato .exe.$'
DADOS    ENDS
CODIGO    SEGMENT
         ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
INICIO:  MOV AX,DADOS
         MOV DS,AX           ; Inicializacao de DS
         MOV AH,09H
         LEA DX,MSG1         ; Aponta para o texto
         INT 21H             ; Escreve MSG1 na tela
         MOV AH,4CH          ; Termina e retorna ao DOS
         INT 21H
CODIGO    ENDS
         END INICIO
```

Versão .com

```
CODIGO    SEGMENT
         ASSUME CS:CODIGO, DS:CODIGO
         ORG 100H
INICIO:  MOV AH,09H
         LEA DX, MSG1        ; Aponta para o texto
         INT 21H             ; Escreve MSG1 na tela
         MOV AH,4CH          ; Termina e retorna ao DOS
         INT 21H
MSG1     DB 'Este programa tem formato .com.$'
CODIGO    ENDS
         END INICIO
```


Anexo 6 – A diretiva ASSUME

A diretiva ASSUME associa um registrador de segmento a um segmento determinado. Esta associação pode parecer, a princípio, desnecessária, mas é extremamente útil em situações como a explicada a seguir.

Suponha um programa que utilize os dois segmentos de dados apresentados abaixo, e que pretenda endereçar DADOS1 via DS e DADOS2 via ES.

```
DADOS1  SEGMENT
X1      DB 01
Y1      DB 01
Z1      DB 01
DADOS1  ENDS

DADOS2  SEGMENT
X2      DB 02
Y2      DB 02
Z2      DB 02
DADOS2  ENDS
```

O grande benefício da diretiva ASSUME está em dispensar o programador de escrever o prefixo de modificação de segmento (segment override prefix) nas instruções que referenciam dados fora do segmento default.

Se isto não fosse assim, então as instruções de acesso às variáveis X2, Y2 e Z2 precisariam incluir o prefixo de modificação de segmento (ES:), como por exemplo em

```
MOV AL,ES:X2
```

que o assembler construiria como

```
MOV AL,ES:[0000H]
```

em que 0000H é o offset da variável X2.

Esta abordagem teria duas grandes desvantagens:

- se o prefixo fosse esquecido, o assembler ainda utilizaria o offset da variável X2, mas geraria o código

```
MOV AL,[0000H]
```

e acabaria endereçando a variável X1;
- se a variável X2 precisasse ser mudada para o segmento DADOS1, então o programador teria que percorrer a listagem e retirar o prefixo de modificação de segmento de todas as instruções que referenciassem X2; da mesma forma, uma variável passada do segmento DADOS1 para o segmento DADOS2 teria que receber o prefixo em todas as instruções que a referenciassem.

Com a diretiva ASSUME, porém, não é necessário utilizar o prefixo de modificação de segmento quando se referencia uma variável pelo nome. O assembler procura o nome da variável nos segmentos e gera o código com o prefixo de modificação de segmento se necessário, mesmo que este não apareça no código-fonte.

Em termos do exemplo anterior, isto significa que o código gerado para a instrução

```
MOV AL,Y2
```

seria

```
MOV AL,ES:[0001].
```

Mais ainda, se Y2 fosse movida para o segmento DADOS1, o assembler passaria a gerar o código correto sem precisar de qualquer alteração no programa-fonte.

A título de exercício, experimente montar o programa a seguir, gerando a listagem (LST). Nesta listagem, observe que, ao referenciar as variáveis do segmento DADOS2 (para o qual vale ASSUME ES), o montador inclui automaticamente o prefixo de modificação de segmento, reconhecível pelo código hexadecimal 26H. Experimente retirar a diretiva ou mudar variáveis de um segmento outro, acompanhando o que o assembler faz.

```
PILHA    SEGMENT STACK
          DB 128 DUP (?)
PILHA    ENDS
DADOS1   SEGMENT
X1        DB 01
Y1        DB 01
Z1        DB 01
DADOS1   ENDS
```

```
DADOS2  SEGMENT
X2      DB 02
Y2      DB 02
Z2      DB 02
DADOS2  ENDS
CODIGO  SEGMENT
        ASSUME CS:CODIGO, SS:PILHA, DS:DADOS1, ES:DADOS2
START:  MOV AX,DADOS1
        MOV DS,AX
        MOV AX,DADOS2
        MOV ES,AX
        MOV X1,11H
        MOV Y1,11H
        MOV Z1,11H
        MOV X2,22H
        MOV Y2,22H
        MOV Z2,22H
        MOV AH,4CH      ; TERMINAR
        INT 21H
CODIGO  ENDS
        END START
```


Anexo 7 – Programas com múltiplos arquivos

Quando se trabalha com projetos maiores, desenvolvidos em equipe, é interessante distribuir as tarefas de implementação de código. Para tanto, as ferramentas de software (assembler, linker) devem ser capazes de:

- montar separadamente diferentes arquivos-fonte, que referenciem dados e sub-rotinas uns dos outros;
- reunir os arquivos-objeto gerados desta forma num único arquivo executável.

Existem aqui dois problemas, cuja solução é descrita a seguir.

O primeiro problema aparece quando um arquivo-fonte referencia um símbolo declarado em outro arquivo. Neste caso, se nada for feito, o assembler gera uma mensagem de erro, reclamando do símbolo desconhecido. Para evitar que este erro seja gerado, utiliza-se a diretiva EXTRN, que diz ao assembler para ignorar o erro, prometendo-lhe que o linker encontrará o símbolo mais tarde. A sintaxe da diretiva é

EXTRN <nome do símbolo externo>: <tipo do símbolo externo>

Por exemplo, se um arquivo-fonte contém a linha

CALL FAR PTR SHOWMSG

e SHOWMSG é uma sub-rotina definida em outro arquivo, então a diretiva

EXTRN SHOWMSG: FAR

deve ser colocada no início do segmento de código que contém o CALL. O assembler gerará então o código correspondente a CALL 0000:0000, cabendo ao linker corrigir o endereço de chamada quando reunir os diferentes arquivos-objeto.

O segundo problema que tem que ser resolvido é que, por default, o linker não tem acesso aos símbolos declarados dentro de um arquivo-fonte. Portanto, no

exemplo acima, não basta que o arquivo que chama a sub-rotina SHOWMSG seja linkado com o arquivo-objeto que contém sua implementação. É preciso dizer ao assembler, já na geração do código objeto, que divulgue o símbolo SHOWMSG, para que o linker possa encontrá-lo mais tarde. Isto se faz colocando a diretiva

PUBLIC SHOWMSG

no início do arquivo-fonte.

As listagens a seguir ilustram o que foi dito.

O módulo principal

Este módulo chama a sub-rotina SHOWMSG, definida em outro módulo. Por isso, precisa da diretiva EXTRN.

```
PILHA    SEGMENT STACK
          DW 80H DUP(?)
PILHA    ENDS
DADOS    SEGMENT
MSG1     DB 'Foi chamada uma rotina de outro
segmento.$',0DH,0AH
DADOS    ENDS
CODE     SEGMENT
          ASSUME CS:CODE,DS:DADOS,SS:PILHA
EXTRN    SHOWMSG:FAR           ; AQUI!!
START:   MOV AX,DADOS
          MOV DS,AX
          LEA AX,MSG1
          PUSH AX
          CALL FAR PTR SHOWMSG
          ADD SP,2
          MOV AH,4CH
          INT 21H
CODE     ENDS
          END START
```

O módulo com a sub-rotina auxiliar

Este módulo define a sub-rotina SHOWMSG, chamada em outro módulo. Por isso, precisa exportá-la com a diretiva PUBLIC.

```
PUBLIC  SHOWMSG                ; AQUI!!
AUXCODE SEGMENT
    ASSUME CS:AUXCODE
SHOWMSG PROC FAR
    PUSH BP
    MOV BP,SP
    MOV AH,09H
    MOV DX,[BP+6]
    INT 21H
    POP BP
    RET
SHOWMSG ENDP
AUXCODE ENDS
END
```

Para criar o programa executável, primeiro se montam os módulos com o assembler, um de cada vez. Em seguida, chama-se o linker passando-lhe como parâmetros os nomes dos dois arquivos-objeto (a extensão não é necessária). Por default, o nome do arquivo executável será o mesmo do primeiro arquivo-objeto passado.

Por exemplo, se os módulos acima forem salvos como Test.asm e ShowMsg.asm, respectivamente, a geração do executável Test.exe se faz com:

```
MASM TEST,,,,
MASM SHOWMSG,,,,
LINK TEST SHOWMSG,,,,
```


Anexo 8 – Divisão de números inteiros

A divisão de números inteiros em programas escritos em Assembly é um capítulo extenso, que não será abordado em sua totalidade aqui. No entanto, é muito instrutivo aprender ao menos como se faz esta operação em alguns casos. A dificuldade associada a esta operação é freqüentemente subestimada, visto que o leigo assume que a instrução DIV do processador resolverá o problema. Veremos a seguir que isto não é assim tão simples.

Embora o material apresentado aqui seja ilustrado com código para o 8086, os princípios são gerais e podem ser aplicados a outros processadores.

O conjunto de instruções do 8086 inclui a instrução DIV, que se comporta de duas formas distintas, dependendo de o operando utilizado ser um registrador de 8 ou de 16 bits:

- DIV <reg8>: divide AX pelo reg8 e coloca o quociente em AL e o resto em AH;
- DIV <reg16>: divide DX:AX pelo reg16 e coloca o quociente em AX e o resto em DX.

A princípio, pode parecer que, com a segunda alternativa, é possível dividir qualquer número de 32 bits por outro. De fato, é possível dividir números de 32 bits por números de 16 bits, colocando os words que compõem o divisor em DX e AX e o dividendo, por exemplo, em BX e executando então a instrução DIV BX.

No entanto, é importante considerar que esta divisão só terá sucesso se o quociente for tal que caiba em AX, e portanto menor do que 10000H. Caso isto não aconteça, o processador gera automaticamente uma interrupção de software (INT 00H), acusando *overflow* de divisão. Por exemplo, a instrução citada pode ser utilizada para dividir 12345678H por 2000H (o quociente é

91A2H e o resto 1678H), mas não para dividir 12345678H por 0002H, cujo quociente seria 091A2B3CH.

O que foi dito acima coloca, portanto, o seguinte problema: como utilizar da maneira mais eficiente a instrução DIV para fazer a divisão de qualquer número de 32 bits por qualquer número de 16 bits? O texto que segue apresenta um algoritmo com essa finalidade.

Seja D um número de 32 bits, que deve ser dividido por d , um número de 16 bits, ambos inteiros sem sinal. Então estamos interessados em calcular o quociente e o resto da divisão

$$\frac{D}{d}. \quad (8.1)$$

Note que o número D pode ser escrito na forma

$$D = 10000H \times D_1 + D_0, \quad (8.2)$$

em que D_1 e D_0 são, respectivamente, os words mais e menos significativos de D .

Note também que, com a instrução DIV, sempre será possível dividir dois números de 16 bits, bastando para isso fazer $DX=0000H$ e colocar o dividendo em AX . Por isso, é possível fazer a divisão

$$\frac{D_1}{d}$$

e escrever

$$D_1 = q_1 d + r_1, \quad (8.3)$$

em que q_1 e r_1 são, respectivamente, o quociente e o resto dessa divisão. Colocando 8.2 e 8.3 em 8.1, vem:

$$\frac{D}{d} = 10000H \times q_1 + \frac{10000H \times r_1 + D_0}{d}. \quad (8.4)$$

A equação 8.4 representa o primeiro passo da divisão. Em particular, está nos dizendo que, se for possível fazer a divisão

$$\frac{10000H \times r_1 + D_0}{d} \quad (8.5)$$

e se o resultado desta jamais passar de FFFFH, então q_1 é o word mais significativo do resultado procurado.

Veremos agora que é exatamente isto que acontece, porque a divisão 8.5 nunca pode causar overflow. Para tanto, considere que o maior valor que r_1 pode ter é $d - 1$, e que o maior valor que D_0 pode ter é FFFFH. Então

$$\frac{10000H \times r_1 + D_0}{d} \leq \frac{10000H \times (d - 1) + FFFFH}{d}, \quad (8.6)$$

que pode ser trabalhada para se chegar a

$$\frac{10000H \times r_1 + D_0}{d} \leq FFFFH + \frac{d - 1}{d}. \quad (8.7)$$

A equação 8.7 nos diz que o maior quociente que pode ser obtido como resultado de (8.5) é FFFFH, um número de 16 bits, e que maior resto é $d - 1$, também de 16 bits. Por isso, o cálculo de (8.5) pode ser feito pela instrução DIV, colocando-se r_1 em DX e D_0 em AX. Esta divisão permite escrever o resultado de (8.5) como

$$\frac{10000H \times r_1 + D_0}{d} = q_0 d + r_0, \quad (8.8)$$

em que q_0 e r_0 são o quociente e o resto da divisão, respectivamente.

Colocando 8.8 em 8.4, vem:

$$\frac{D}{d} = 10000H \times q_1 + q_0 + \frac{r_0}{d}, \quad (8.9)$$

que significa que a divisão tem por quociente o número de 32 bits formado pelos words q_1 e q_0 e que seu resto é r_0 .

A equação 8.9 permite então escrever o seguinte algoritmo para fazer a divisão pretendida:

1. fazer DX = 0000 e AX = D_1 ;
2. fazer BX = d ;
3. executar DIV BX, que coloca q_1 em AX e r_1 em DX;
4. salvar q_1 como word mais significativo do quociente;
5. colocar D_0 em AX (DX permanece com r_1);
6. executar novamente DIV BX, que coloca q_0 em AX e r_0 em DX;
7. salvar q_0 como word menos significativo do quociente;
8. salvar r_0 como resto.

O procedimento apresentado pode ser generalizado para dividendos maiores do que 32 bits. Basta começar com DX = 0000 e AX = word mais significativo e repetir a divisão até chegar ao word menos significativo.

Referências bibliográficas

Conceitos básicos de Sistemas Digitais

- [Mors88] Morse, Stephen P.: *Microprocessadores 8086/8088 Arquitetura, projeto, sistemas e programação*. Editora Campus, Rio de Janeiro (1988)
- [WS99] Wakerly, John F.; Stone, Harold Samuel: *Digital Design: Principles and Practices*. Prentice Hall (1999) ISBN 0137691912

Microprocessador 8085

- [Inte77] Intel Corporation: *8080/8085 Assembly Language Programming*. Manual order number: 9800940
- [Visc82] Visconti, Antônio Carlos: *Microprocessadores 8080 e 8085 – Vol. 2: Software*. Editora Érica Ltda (1982)

Microprocessador 8086

- [BK91] Brown, Ralf; Kyle, Jim: *PC Interrupts*. Addison-Wesley (1991) ISBN 0-201-57797-6
- [NAW93] Norton, P.; Aitken, P; Wilton, R.: *A Bíblia do Programador*. Editora Campus (1993) ISBN 85-7001-859-2
- [Wake81] Wakerly, John F.: *Microcomputer Architecture and Programming*. John Wiley & Sons, NY (1981) ISBN 0471052310
- [ZM99] Zelenovsky, R.; Mendonça, A.: *PC: Um guia prático de hardware e interfaceamento*. MZ Editora Ltda (1999)

Consulte também o item [Links Interessantes](#), na homepage que dá suporte a este livro, para saber como obter livros gratuitos da Internet.

Outros livros de professores do EEL / UFSC

Geraldo Kindermann

☎ +48 331-9731, 📠 +48 331-7538, ✉ geraldo@gpse.ufsc.br

📖 Curto-circuito

📖 Descargas Atmosféricas

📖 Choque Elétrico

📖 Proteção de Sistemas Elétricos de Potência

Geraldo Kindermann / Jorge Mario Campagnolo

📖 Aterramento Elétrico

J.P. Assumpção Bastos

✉ jpab@grucad.ufsc.br

📖 Eletromagnetismo e Cálculo de Campos. Editora da UFSC,
terceira edição em 1996

Sidnei Noceti Filho

✉ sidnei@linse.ufsc.br

📖 Filtros Seletores de Sinais. Editora da UFSC.