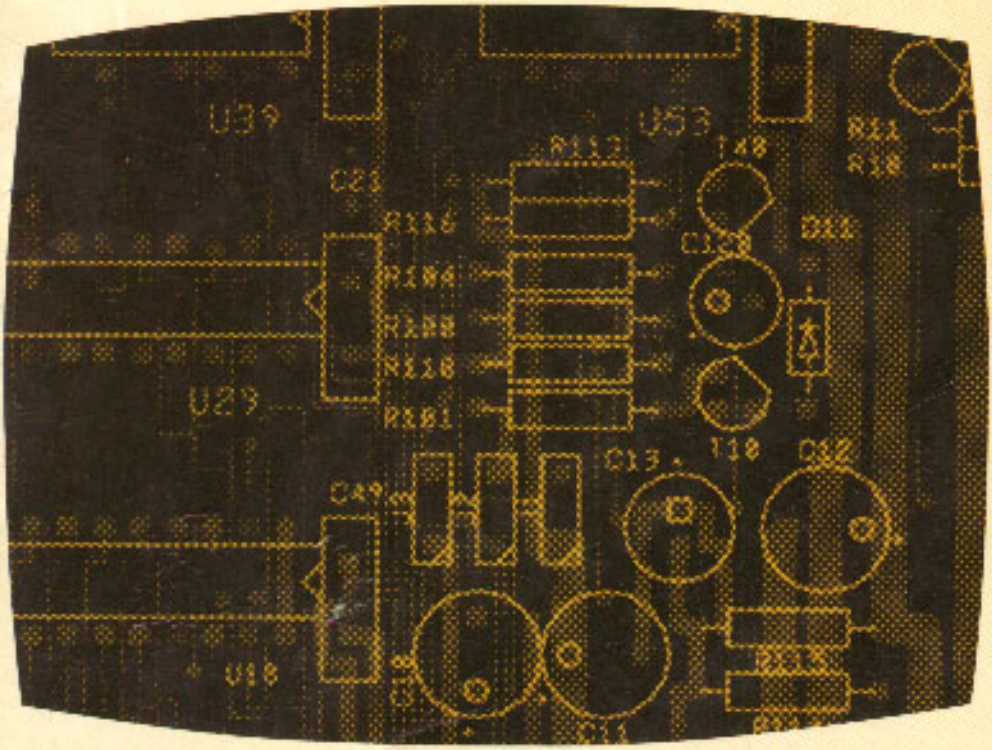
linguagem de máquina

ASSEMBLY Z-8Ø

MSX



Editora Aleph

Rossini – Figueredo

linguagem de máquina

MSX

AGRADECIMENTOS

os Autores gostariam de agradecer o prof. Ricardo Feltre da Editora Moderna, o Eng. Helio Fromer pelo suporte técnico, Milton Maldonado Jr. pelo pro-programa Monitor, o Eng. Moris Arditti da Gradiente pelo apoio, a Sueli da FAU, Fernanda Chaui Petroni pela "força" e aos Beatles pela inspiração do Flávio e ao Sisters of Mercy pela do Henrique.

Rossini – Figueredo

linguagem de máquina MSX

1ª edição
1987

Editora Aleph

EXPEDIENTE

Coordenação Editorial - PIERLUIGI PIAZZI
Coordenação Didática - BETTY FROMER PIAZZI
Produção Editorial - ROSA KOGAN FROMER
Editoração - RENATO DA SILVA OLIVEIRA
LUIZ TARCÍSIO DE CARVALHO JR
Aberturas - ODILON D. NICOLETTI
Ilustrações e Arte - ANA LÚCIA ANTICO

ALEPH
Publicações e Assessoria Pedagógica Ltda
Caixa Postal 20.707
CEP 01498 São Paulo SP
Tel: (011) 843-3202

Dados de Catalogação na Publicação (CIP) Internacional
(Câmara Brasileira do Livro, SP, Brasil)

Rossini, Flávio, 1959-
R744L Linguagem de máquina : MSX / Rossini-Figueredo. -- São Paulo : Aleph, 1987.
(Série didática)

1. Linguagens de programação (Computadores)
2. Microcomputadores - Programação 3. MSX (Computadores) - Programação I. Luz, Henrique de Figueredo, 1967- II. Título. III. Série.

87-2035 CDD-001.6424
-001.642

Índices para catálogo sistemático:

1. Microcomputadores : Linguagem de programação : Processamento de dados 001.6424
2. MSX : Computadores : Programação : Processamento de dados 001.642
3. MSX : Linguagem de programação : Computadores : Processamento de dados 001.6424

SUMÁRIO

NOTA DO EDITOR

O Brasil é realmente um país "sui generis". Enquanto que, de um lado, temos uma política estúpida na área de informática, que só estimula a pirataria e a picaretagem, do outro lado temos uma multidão de usuários extremamente inteligentes que, na base do auto-didatismo, adquiriu um nível de conhecimentos inegável. Este time de auto-didatas leva adiante o desenvolvimento do país na área de micro-computação apesar da política oficial (melhor dizendo, "à revelia") que parece escolher a dedo as medidas a serem tomadas para desestimular o desenvolvimento tecnológico do Brasil, numa área tão crítica e tão carente.

Para atender estes auto-didatas, sedentos de informações, estamos lançando esta obra que consiste na reelaboração, ampliação e adaptação de um dos best-sellers da micro-computação, a bíblia dos sinclairistas escrita originalmente pelo FLAVIO ROSSINI. Nesta nova versão foram incluídas as instruções secretas do Z-80 (ausentes até no próprio manual dos fabricantes do chip) e configuramos o livro para o "herdeiro" do SINCLAIR entre os usuários inteligentes: o MSX.

Esperamos sinceramente que esta obra possa significar mais um impulso no desenvolvimento do único hardware que precisa ser estimulado: o cérebro do usuário.

Certamente, a leitura deste livro não será das mais fáceis, mesmo para programadores com muita experiência no BASIC. Uma nova linguagem será ensinada e além da parte lógica, que constitui propriamente a programação, será necessário a aquisição de um novo vocabulário e de novas notações. Um exemplo disso é o uso frequente dos sistemas de numeração hexadecimal e binário. Se você não está familiarizado com eles, leia antes o apêndice 1.

Carregados de razão são os que dizem que o Brasil só vai para frente da meia noite as seis da manhã, quando os políticos e os marajás estão dormindo.

Acrescentaríamos: "e quando os apaixonados por micros estão sentados à frente de suas máquinas"!

1

CONHECENDO MELHOR O Z-8Ø

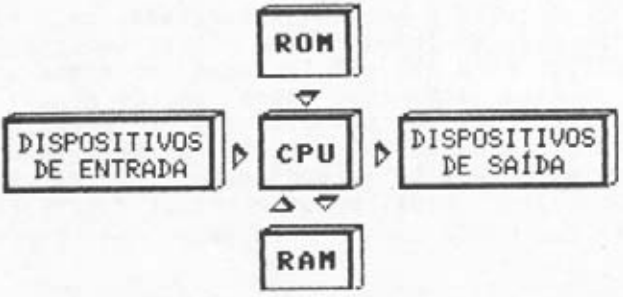
Um microcomputador consta de duas "partes" principais: Hardware e Software. O Hardware é formado pelos circuitos eletrônicos internos e periféricos (impressora, drive, plotter, etc.). O Software é formado pelos programas (Sistema Operacional, compilador, programas fonte) que são executados pelo hardware.

Vamos apresentar, neste capítulo, as noções sobre o sistema MSX e as características do Z-8Ø.

Não é nossa intenção fazer uma descrição minuciosa de toda a estrutura do microprocessador, mas mostrar-lhe que o Z-8Ø não é um bicho de 7 cabeças. Durante a leitura deste livro você verá que a Linguagem de Máquina não é algo reservado somente aos chamados "gênios da computação".

Vejamos na, figura 1.1, a estrutura básica de todo microcomputador.

FIGURA 1.1 - Estrutura básica de um microcomputador.



Ou seja, para um microcomputador funcionar ele precisa, no mínimo, das seguintes partes:

* CPU - Unidade Central de Processamento. Circuito capaz de obedecer a uma série de instruções armazenadas na memória e de enviar e receber informações de outros circuitos.

* ROM - Circuito que contém préviamente gravado o Sistema Operacional e o Interpretador Basic.

* RAM - Memória volátil de leitura e escrita. Nela são armazenados os programas, variáveis, variáveis de sistema e buffer de arquivos.

* Dispositivos de Entrada - Qualquer meio que forneça dados para a CPU processar (teclado, disk drive, joystick, sensores, mouse, light pen, etc.)

* Dispositivo de Saída - Qualquer meio que receba as informações processadas mostrando-as ao usuário ou armazenando-as, de acordo com a função do dispositivo.

O CLOCK DO MICROPROCESSADOR

Certamente você já viu, principalmente na propaganda do micro, a característica do MSX de ter um clock de 3.5 MHz.

Mas afinal o que é esse tão falado clock?

O clock é praticamente o "coração" do microcomputador. A cada "tic" do clock, ou seja, a cada 1/3.500.000 segundos, uma instrução (ou parte de uma instrução) em Linguagem de Máquina é executada.

O MICROPROCESSADOR Z-80

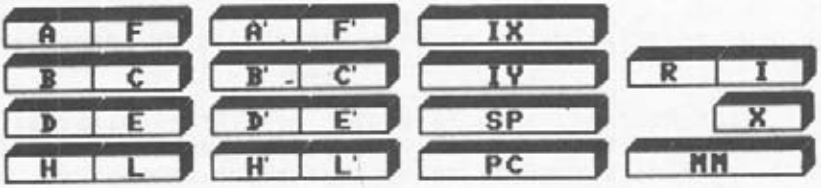
O Z-80, é um microprocessador de 8 bits. Ou seja, ele é capaz de manipular apenas um byte (8 bits) a cada "tic" (1/3.500.000 segundos) do clock.

Para armazenar dados, o Z-80 possui algumas memórias internas que são chamadas registros ou registradores. Alguns registros são "privilegiados", pois existem instruções em Linguagem de Máquina específicas para eles. Cada registro possui 8 bits (ou seja, um byte).

OS REGISTROS DO Z-80

A figura 1.2 apresenta a configuração de registros do Z-80.

FIGURA 1.2 - REGISTROS DO Z-80.



A cada registro é associada uma letra. Porém alguns registros são obrigatoriamente "colados" formando um único registro de 16 bits. Esses registros recebem um nome de duas letras (SP, IX, etc.).

Podemos dividir os registros em dois grandes grupos: principais e auxiliares.

Os registros principais são o X, MM, A, F, PC, SP e I.

X - Registro onde são colocados, a cada "tic" do clock os dados ou instruções que passam do Z-80 para "fora" (memória e/ou periféricos) ou vice-versa; é "invisível" ao usuário mesmo em Linguagem de Máquina, ou seja, nunca pode ser acessado ou modificado diretamente por software.

MM - Registro de 16 bits também inacessível ao usuário. O MM contém sempre o endereço de memória ou periférico a ser acessado a cada "tic" do clock. Como o maior número representável com 16 bits é 65535, o Z-80 só pode acessar 64 Kbytes de memória. No caso de periféricos, somente metade desse registro é ativo. Por isso o Z-80 só pode receber e/ou enviar dados (8 bits por vez) a 256 periféricos.

A - Tem a finalidade básica de armazenar o resultado de uma operação lógica ou aritmética realizada pelo microprocessador. No entanto, pode ser usado como um registro "comum" para armazenar outros dados desde que não seja realizada uma operação lógica ou aritmética.

PC (Program Counter) - Serve como "ponteiro", indicando o endereço da instrução a ser executada pela CPU. O

PC é incrementado após a execução da instrução ou então é alterado para um novo valor com instruções tipo GOTO ou GOSUB do BASIC.

F (Flag) - Registro onde cada bit armazena uma condição de status da CPU, geralmente em referência ao registro A. Por exemplo, se o número no registro A é zero ou não. Ele é fundamental para as várias operações da CPU, principalmente para as instruções de condição (IF).

SP (Stack Pointer) - Ao executar certas instruções, alguns dados têm que ser guardados temporariamente numa região de memória denominada stack (ou pilha). O uso dessa região de memória é no sistema LIFO ((Last In First Out) e será vista em detalhes mais para o final do livro. Esse registrador aponta sempre para o topo da pilha, ou seja, no próximo endereço desta região de memória será inserido um novo valor.

I (Interrupção) - Alguns dispositivos externos trabalham de maneira "independente" e devem acionar rotinas a serem executadas pela CPU no momento em que eles as "requisitarem". Para isso, eles devem interromper o Z-80 e esse registro ajuda a coordenar esse processo. Ele fornece o byte mais significativo do endereço da rotina de interrupção. Maiores detalhes serão vistos no final do livro.

Os registros auxiliares são o B, C, D, E, H, L, B', C', D', E', H', L', A', F', IX, IY, e R.

B, C, D, E, H e L - Registros utilizados em operações gerais. Podem ser utilizados em pares para operações de 16 bits, formando obrigatoriamente os pares BC, DE e HL.

B', C', D', E', H', L', A' e F' - Registros alternativos (série dos "prime") absolutamente similares aos registros B, C, D, E, H, L, A e F. Esses registros não podem ser manipulados diretamente. O que pode ocorrer é a troca de todos os valores dos registros com seu equivalente na série dos "prime".

IX e IY (Indexação) - A CPU utiliza esses registros de 16 bits cada como ponteiros para operações no modo de endereçamento indexado, facilitando a manipulação de tabelas de dados.

R (Refresh) - Este registro é usado para facilitar o processo de "refrescamento" de memória RAM caso se esteja usando memória RAM dinâmica. Esse tipo de RAM é mais barato que a RAM "convencional" (também chamada de estática) mas perde seu conteúdo após um certo tempo mesmo com alimentação elétrica, a menos que seja "refrescada" por um pulso elétrico; desse modo o Z-80 facilita o uso da RAM dinâmica.

A MEMÓRIA DO COMPUTADOR

Embora o Z-80 só possa endereçar 64 Kbytes de memória, o sistema MSX permite que disponhamos de mais memória, embora não possamos acessar toda ela (que pode chegar a 1Mbyte) ao mesmo tempo.

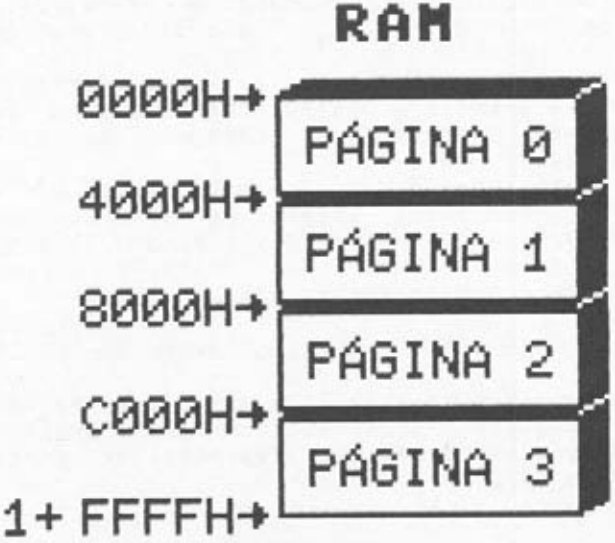
Em sua configuração inicial, um MSX possui 4 slots (2 internos e 2 externos). Um dos slots internos contém a memória RAM (64K) e o outro (slot 0) contém os 32K de RAM.

O padrão MSX não obriga que a memória RAM esteja em um determinado slot. Essa decisão é exclusiva do fabricante.

O Expert 1.1 possui a memória RAM no slot 2 enquanto que o HOTBIT 1.1 a possui no slot 3.

Cada slot é dividido em 4 páginas de 16K cada uma. Observe a figura 1.3.

FIGURA 1.3 - Slot dividido em páginas.



Quando o micro está operando em BASIC, o Z-80 está acessando as páginas 0 e 1 do slot 0, pois é nesse lugar que fica a memória ROM com o BIOS e o interpretador BASIC.

Uma página de memória é sempre fixa, ou seja, se ela for uma página 2, por exemplo, ela não poderá ser acessada como página 0,1 ou 3. Ela possuirá o seu endereçamento fixo entre 8000H e BFFFH.

VRAM - MEMÓRIA PARA VÍDEO

Além dos 64 Kbytes de RAM, existe a memória para vídeo, formada por 16 Kbytes de uso exclusivo do processador de vídeo. Ou seja, o Z-80 não pode acessá-la diretamente como a RAM normal. Qualquer dado deve ser previamente enviado ou solicitado ao processador de vídeo que se encarregará de realizar as operações necessárias.

O PROGRAMA MONITOR

No MSX os primeiros 32 Kbytes não podem ser operados pelo BASIC. Os programas em BASIC são armazenados a partir do endereço 8000H. As Variáveis do Sistema ocupam o final da RAM; e a cada interface ou cartucho instalado perde-se alguns bytes antes das Variáveis do Sistema.

Deste modo ficamos meio "cercados" pelos programas em BASIC ou por buffers de interfaces.

Um bom lugar para se colocar os programas em Linguagem de Máquina é o início da página três. Assim, os programas em BASIC e suas variáveis poderão ocupar até 16 Kbytes e sobrará bastante memória para buffers e variáveis de sistema de um cartucho ou interface. Observe a figura I.4.

Para impedir que o programa BASIC ultrapasse o endereço C000H basta alterar variável de sistema HIMEM (HIghest avaliable MEMory - FC4AH) para o valor desejado ou executar um CLEAR do BASIC.

Com o computador ligado digite:

```
PRINT HEX$(PEEK(&HFC4A)+256*PEEK(&HFC4B))
```

Se nenhuma interface estiver ativa você obterá como resposta o endereço FC38H. Caso contrário o resultado pode ser o mais variado possível, dependendo do número de interfaces usadas.

FIGURA 1.4 - Esquema do slot de RAM de um MSX.

Agora comande:

```
CLEAR 200, &HC000
```

Volte o cursor 3 linhas acima e obtenha o novo valor da variável HIMEM. Como poderia se esperar, devido ao CLEAR executado pelo BASIC, o endereço obtido foi C000H, que será o endereço mais alto no qual o programa BASIC poderá chegar.

Aconselhamos o uso de programas Assemblers e Monitores tipo SIMPLE ASM, MONGIN, HOTASM, ASMGOGAR, etc., o que facilitará em muito o aprendizado da Linguagem de Máquina.

Caso você não possua um, digite o programa monitor da figura 1.5.

FIGURA 1.5 - PROGRAMA MONITOR.

```
1000 CLS:PRINT "Monitor MSX 1986":PRINT
"Milton Maldonado Jr."
1010 PRINT:LINE INPUT ")";A$:IF LEN(A$)
=0 THEN 1010
1020 C$=MID$(A$,1,1):IF C$="L" THEN 113
0 ELSE IF C$="D" THEN 1030 ELSE IF C$="
M" THEN 1060 ELSE BEEP:PRINT "?";:GOTO
1010
1030 E=VAL(MID$(A$,2))
1040 FOR J=1 TO 8:PRINT RIGHT$("0000"+H
EX$(E),4);"  ";:FOR I=0 TO 7:PRINT RIGH
T$("00"+HEX$(PEEK(I+E)),2);" ";:NEXT I:
PRINT:E=E+8:NEXT J
1050 I$=INPUT$(1):IF I$=CHR$(27) THEN 1
010 ELSE PRINT:GOTO 1040
1060 E=VAL(MID$(A$,2))
1070 PRINT:PRINT RIGHT$("0000"+HEX$(E),
4);"  ";RIGHT$("00"+HEX$(PEEK(E)),2);"
";
1080 H$="":FOR I=1 TO 2
1090 I$=INPUT$(1):IF I=1 THEN IF I$=CHR
$(13) THEN E=E+1:GOTO 1070 ELSE IF I$=C
HR$(8) THEN E=E-1:GOTO 1070
1100 IF I$=CHR$(27) THEN PRINT:GOTO 101
0
1110 IF I$)="0" AND I$(="9" OR I$)="A"
AND I$(="F" THEN 1120 ELSE 1090
1120 H$=H$+I$:PRINT I$;:NEXT I:POKE E,V
AL("&H"+H$):E=E+1:GOTO 1070
1130 E=VAL(MID$(A$,2))
1140 RESTORE:FOR I=0 TO 7:READ A$(I),B$
(I):NEXT I:FOR I=0 TO 3:READ C$(I),E$(I
):NEXT I:FOR I=0 TO 7:READ D$(I),F$(I):
NEXT I:READ G$(0),G$(1),G$(2),H$(0),H$(
1),H$(2),I$(0),I$(1),J$(0),J$(1)
1150 FOR K=1 TO 20:C=PEEK(E)
1160 PRINT USING "#####";E;:PRINT "  ";
:GOSUB 1190:IF F)0 THEN I=I+1
1170 PRINT TAB(22);:FOR X=0 TO I-1:I$=H
EX$(PEEK(X+E)):IF LEN(I$)(2 THEN I$="0"
+I$
1180 PRINT I$;" ";:NEXT X:PRINT:E=E+I:N
EXT K:PRINT:I$=INPUT$(1):IF I$=CHR$(27)
 THEN 1010 ELSE 1150
```

```
1190 '-----------------------------
          SEPARA GRUPOS
-----------------------------
1200 F=0:IF C=221 THEN F=1 ELSE IF C=25
3 THEN F=2
1210 C$(2)="HL":IF F=1 THEN C$(2)="IX"
ELSE IF F=2 THEN C$(2)="IY"
1220 IF C=203 THEN 1740 ELSE IF C=237 T
HEN 1800
1230 IF F>0 AND PEEK(E+1)=203 THEN 1740
1240 IF F>0 THEN C=PEEK(E+1)
1250 '-----------------------------
          COMANDOS DIRETOS
-----------------------------
1260 E$(2)=C$(2)
1270 I=1:IF C=39 THEN PRINT "DAA";ELSE
IF C=47 THEN PRINT "CPL";ELSE IF C=249
THEN PRINT "LD SP,";C$(2);ELSE IF C=227
 THEN PRINT "EX (SP),";C$(2);ELSE IF C=
118 THEN PRINT "HALT";ELSE IF C=201 THE
N PRINT "RET";ELSE IF C=31 THEN PRINT "
RRA";ELSE 1290
1280 RETURN
1290 IF C=235 THEN PRINT "EX DE,HL";ELS
E IF C=8 THEN PRINT "EX AF,AF'";ELSE IF
 C=217 THEN PRINT "EXX";ELSE IF C=233 T
HEN PRINT "JP (";C$(2);")";ELSE IF C=0
THEN PRINT "NOP";ELSE IF C=243 THEN PRI
NT "DI";ELSE IF C=251 THEN PRINT "EI";E
LSE 1310
1300 RETURN
1310 IF C=7 THEN PRINT "RLCA";ELSE IF C
=15 THEN PRINT "RRCA";ELSE IF C=23 THEN
 PRINT "RLA";ELSE IF C=55 THEN PRINT "S
CF";ELSE IF C=63 THEN PRINT "CCF";ELSE
1330
1320 RETURN
1330 I=2:IF C=211 THEN PRINT "OUT (";:G
OSUB 1610:PRINT "),A";ELSE IF C=219 THE
N PRINT "IN A,(";:GOSUB 1610:PRINT")";E
LSE 1350
1340 RETURN
1350 IF C=195 THEN PRINT "JP ";ELSE IF
C=205 THEN PRINT "CALL ";ELSE 1380
1360 PRINT MID$(STR$(PEEK (E+1)+256*PEE
K(E+2)),2,5);:I=3:RETURN
1370 RETURN
1380 IF C=16 THEN PRINT "DJNZ ";ELSE IF
 C=24 THEN PRINT "JR ";ELSE 1410
```

```
1390 X=PEEK(E+1):IF X<128 THEN Y=E+X+2
ELSE Y=E+X-254
1400 PRINT MID$(STR$(Y),2,5);:I=2:RETUR
N
1410 IF C>127 AND C<192 THEN X=C\8-16:Y
=C MOD 8:PRINT A$(X);:IF X=0 OR X=1 OR
X=3 THEN PRINT " A,";ELSE PRINT " ";ELS
E 1430
1420 PRINT B$(Y);:I=1:RETURN
1430 IF C\64>0 THEN 1570
1440 IF C=34 THEN E=E+SGN(F):PRINT "LD
(";:GOSUB 1360:PRINT "),";C$(2);:ELSE I
F C=42 THEN E=E+SGN(F):PRINT "LD ";C$(2
);",(";:GOSUB 1360:PRINT ")";:ELSE 1460
1450 E=E-SGN(F):RETURN
1460 IF C=50 THEN PRINT "LD (";:GOSUB 1
360:PRINT "),A";ELSE IF C=58 THEN PRINT
 "LD A,(";:GOSUB 1360:PRINT ")";:RETURN
 ELSE 1480
1470 I=3:RETURN
1480 X=C MOD 8:IF X=4 THEN PRINT"INC";E
LSE IF X=5 THEN PRINT"DEC";ELSE 1500
1490 Y=C\8:PRINT" ";B$(Y);:I=1:RETURN
1500 X=C MOD 16:IF X=3 THEN PRINT"INC "
;ELSE IF X=11 THEN PRINT"DEC ";ELSE IF
X=9 THEN PRINT"ADD ";C$(2);",";ELSE 152
0
1510 X=C\16:PRINTC$(X);:I=1:RETURN
1520 IF C MOD 8=6 AND F=0 THEN PRINT "L
D ";B$(C\8);",";:GOTO 1610
1530 IF C MOD 16=1 THEN E=E+SGN(F):PRIN
T"LD ";C$(C\16);",";:GOSUB 1360 ELSE 15
50
1540 E=E-SGN(F):RETURN
1550 IF C MOD 16=2 THEN PRINT "LD (";C$
(C\16);"),A";ELSE IF C MOD 16=10 THEN P
RINT "LD A,(";C$(C\16);")";ELSE 1570
1560 I=1:RETURN
1570 IF C\32=1 AND C MOD 8=0 THEN PRINT
 "JR ";D$(C\8-4);",";:GOTO 1390
1580 IF C\64<>3 THEN 1680
1590 IF C MOD 8<>6 THEN 1620
1600 I=2:X=C\8-24:PRINTA$(X);" ";:IF X<
2 OR X=3 THEN PRINT"A,";
1610 PRINT MID$(STR$(PEEK(E+1)),2,3);:R
ETURN
1620 X=C MOD 8:IF X=1 THEN PRINT "POP "
;E$(C\16-12);ELSE IF X=5 THEN PRINT "PU
SH ";E$(C\16-12);
```

```
1630 IF X=1 OR X=5 THEN I=1:RETURN
1640 IF X=7 THEN PRINT"RST ";HEX$(C-199
);"H";:I=1:RETURN
1650 IF X=0 THEN PRINT "RET ";D$(C\8-24
);:I=1:RETURN
1660 IF X=2 THEN PRINT "JP ";ELSE PRINT
"CALL ";
1670 PRINT D$(C\8-24);",";:GOTO 1360
1680 IF F=0 AND C\64=1 THEN X=(C-64)\8:
PRINT "LD ";B$(X);:X=(C-64) MOD 8:PRINT
",";B$(X);:I=1:RETURN
1690 I=2:C=PEEK(E+1):IF F>0 AND C MOD 8
=6 AND C>63 THEN PRINT "LD ";B$(C\8-8);
",(";C$(2);"+";:E=E+1:GOSUB 1610:PRINT
")"; ELSE 1710
1700 E=E-1:RETURN
1710 IF C\8=14 THEN E=E+1:PRINT "LD (";
C$(2);"+";:GOSUB 1610:PRINT "),";B$(C-1
12); ELSE 1730
1720 GOTO 1700
1730 I=3:IF C=54 THEN PRINT "LD (";C$(2
);"+";:E=E+1:GOSUB 1610:PRINT "),";:E=E
+1:GOSUB 1610:E=E-1:GOTO 1720 ELSE 1960
1740 '-----------------------------------
          COMANDOS APOS CBH          -
-----------------------------------
1750 Z=1:IF F>0 THEN Z=3
1760 I=2:C=PEEK(E+Z):IF C<64 AND F=0 TH
EN PRINT F$(C\8);" ";B$(C MOD 8);:RETUR
N
1770 I=3:IF C<64 AND F>0 AND (C-6) MOD
8=0 THEN PRINT F$(C\8);" (";C$(2);"+";:
E=E+1:GOSUB 1610:PRINT")";:E=E-1:RETURN
1780 C=PEEK(E+Z):IF F=0 THEN PRINTG$(C\
64-1);" ";CHR$(48+(C\8) MOD 8);",";B$(C
 MOD 8);:I=2:RETURN
1790 IF (C-6) MOD 8=0 THEN PRINTG$(C\64
-1);" ";CHR$(48+(C\8) MOD 8);",(";C$(2)
;"+";:E=E+1:GOSUB 1610:PRINT")";:E=E-1:
RETURN ELSE 1960
1800 '-----------------------------------
          COMANDOS APOS EDH          -
-----------------------------------
1810 I=2:C=PEEK(E+1):IF C<64 OR C=221 O
R C=253 THEN 1960 ELSE IF C>187 THEN 19
20
```

```
1820 IF C=70 THEN PRINT "IM 0";ELSE IF
C=86 THEN PRINT "IM 1";ELSE IF C=94 THE
N PRINT "IM 2";ELSE IF C=77 THEN PRINT
"RETI";ELSE IF C=69 THEN PRINT "RETN";E
LSE IF C=103 THEN PRINT "RRD";ELSE IF C
=111 THEN PRINT "RLD";ELSE 1840
1830 RETURN
1840 IF C=71 THEN PRINT "LD I,A";ELSE I
F C=79 THEN PRINT "LD R,A";ELSE IF C=87
 THEN PRINT "LD A,I";ELSE IF C=95 THEN
PRINT "LD A,R";ELSE 1860
1850 RETURN
1860 H$(3)="OUT":IF C>175 THEN H$(3)="O
T"
1870 IF C>159 AND C MOD 8<4 THEN PRINTH
$(C MOD 4);I$((C MOD 16)\8);J$((C-160)\
16);:RETURN
1880 IF C\64>1 THEN 1960 ELSE IF C MOD
8>1 THEN 1920
1890 IF C=112 OR C=113 THEN 1960
1900 IF C MOD 8=0 THEN PRINT "IN ";B$(C
\8-8);",(C)";ELSE PRINT "OUT (C),";B$(C
\8-8);
1910 RETURN
1920 X=C MOD 16:Y=C\16-4:IF X=10 THEN P
RINT"ADC HL,";C$(Y);ELSE IF X=2 THEN PR
INT "SBC HL,";C$(Y);ELSE 1940
1930 RETURN
1940 IF X=3 THEN PRINT "LD (";:E=E+1:GO
SUB 1360:E=E-1:PRINT "),";C$(C\16-4);EL
SE IF X=11 THEN PRINT "LD ";C$(C\16-4);
",(";:E=E+1:GOSUB 1360:E=E-1:PRINT")";E
LSE 1960
1950 I=4:RETURN
1960 PRINT "Z80 ?";:RETURN
1970 '-------------------------------
          DADOS DA MATRIZ ALFA        -
---------------------------------------
1980 DATA ADD,B,ADC,C,SUB,D,SBC,E,AND,H
,XOR,L,OR,(HL),CP,A
1990 DATA BC,BC,DE,DE,HL,HL,SP,AF
2000 DATA NZ,RLC,Z,RRC,NC,RL,C,RR,PO,SL
A,PE,SRA,P,SLI,M,SRL
2010 DATA BIT,RES,SET
2020 DATA LD,CP,IN,I,D, ,R
```

O monitor possui 3 comandos D, M e L. O comando D realiza um dump de memória. A sua sintaxe é a seguinte:

D xxxx

onde xxxx é o endereço inicial do dump. Se o endereço estiver em hexadecimal, deverá ser precedido pelos caracteres "&H".

O comando M permite a alteração de bytes da memória, possuindo a seguinte sintaxe:

M xxxx

onde xxxx é o endereço do primeiro byte a ser alterado. Se o endereço for dado em hexadecimal ele deverá ser precedido por "&H".

Quando não for desejada a alteração de um byte, deve ser pressionado RETURN.

O valor a ser alterado deve estar sempre em hexadecimal.

O comando L realiza um disassembler dos códigos contidos na memória. Sua sintaxe é a seguinte:

L xxxx

onde xxxx é o endereço inicial a ser disassemblado, em decimal.

Para encerrar qualquer comando, basta pressionar ESC e o cursor do monitor voltará à tela.

Experimente, então, colocar alguns números hexadecimais na memória do micro, a partir do endereço C000H. Lembre-se que cada endereço da memória corresponde a um byte de dados.

Execute o monitor com RUN e digite:

```
>M &HC000

C000   00   A4
C001   00   B7
C002   00   04
C003   00   12
C004   00   06
C005   00   71
C006   00   FD
C007   00   A1
C008   00   00
C009   00   CE
C00A   00
```

A seguir, digite ESC.

Agora, para verificar se o comando M funcionou corretamente, digite:

```
D &HC000
```

Se você digitou tudo corretamente, os dados da figura 1.6 deverão aparecer na tela.

FIGURA 1.6 - DUMP DO PROGRAMA MONITOR.

```
>D &HC000
C000   A4 B7 04 12 06 71 FD A1
C008   00 CE 00 00 00 00 00 00
C010   00 00 00 00 00 00 00 00
C018   00 00 00 00 00 00 00 00
C020   00 00 00 00 00 00 00 00
C028   00 00 00 00 00 00 00 00
```

Note que estamos apenas colocando os códigos hexadecimais na memória e, mesmo que esses códigos (A4H, B7H, etc.) significassem um programa em Linguagem de Máquina, providências especiais seriam necessárias para executar o programa.

Esses códigos podem representar um programa em Linguagem e Máquina colocado a partir do endereço C000H. Por exemplo, o código A4H pode significar "compare o resultado com o próximo byte (B7H)" e assim por diante...

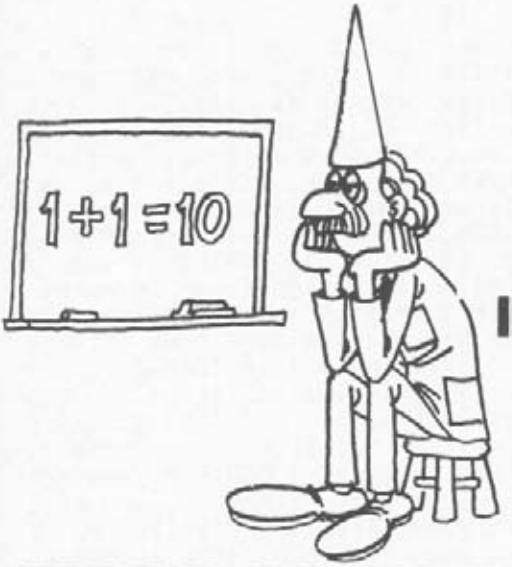
Note que não existem números de linhas de programa como em BASIC, mas as instruções são executadas na sequência em que estão na memória.

A grande vantagem da Linguagem de Máquina está na grande velocidade e em permitir uma maior visualização e controle sobre o micro. Porém as suas instruções são mais simples e limitadas e exigem um maior cuidado na hora da programação.

RESUMO

Vimos, neste capítulo, como são as instruções em Linguagem de Máquina (sempre em grupos de 2 dígitos), a disponibilidade de memória no MSX e onde colocar os programas em Linguagem de Máquina.

Foi apresentado o programa MONITOR (em BASIC) que permite colocar programas elaborados em códigos hexadecimais na memória RAM, verificar o conteúdo da memória e disassemblar os bytes da memória.



2

PRIMEIRAS INSTRUÇÕES

REGISTROS INTERNOS DE 8 BITS

Iniciaremos o capítulo explorando o conceito de "registro", termo empregado propositadamente na capítulo anterior quando falamos sobre a memória. Um registro é um circuito eletrônico capaz de memorizar bits (oito, no nosso caso), podendo representar e armazenar números de 0 a 255 (00H a FFH).

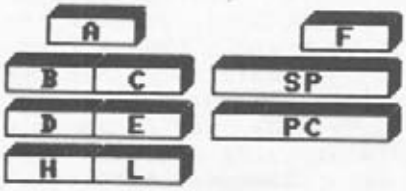
O microprocessador do MSX (Z-80) possui vários registros internos muito utilizados nos programas em Linguagem de Máquina, dos quais sete serão estudados neste capítulo. Estes registros não fazem parte da memória RAM e são chamados, respectivamente de A, B, C, D, E, H e L. Alguns deles têm a propriedade de "se juntarem", quando necessário, formando um único registro de 16 bits; assim, podemos formar os pares de registros BC, DE, HL (só estes pares são possíveis).

Por exemplo, se o registro B contém o número 1AH (160 em decimal) e o registro C contém B2H (178 em decimal), o par BC contém o número 1AB2H (256 * 160 + 178 = 41138 em decimal).

Como vimos no capítulo anterior, as instruções em Linguagem de Máquina são representadas por códigos de dois bytes cada. Isto é feito para evitar a confusão que poderia ser criada usando apenas os números 0 e 1. Mesmo assim, temos agora apenas 16 símbolos disponíveis (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E e F) e, para programas razoavelmente grandes, mesmo

isto pode trazer confusão. Dessa forma, para facilitar a compreensão do ponto de vista humano, costuma-se associar, a cada instrução, grupos de letras em forma "mnemônica" para auxiliar a programação, chamamos "códigos de operação", que têm como objetivo associar uma "ordem" a cada grupo de letras. Por exemplo, se o código 10100111 significar "some", seu correspondente em hexadecimal seria A7H e seu mnemônico poderia ser ADD. Assim, dependendo da instrução em Linguagem de Máquina, cada mnemônico poderá significar um ou mais grupos de 2 dígitos hexadecimais. Dessa maneira, para programar em Linguagem de Máquina deveremos aprender os mnemônicos correspondentes a cada instrução do microprocessador e fazer o programa utilizando os mesmos; no final, com o auxílio de uma tabela ou um ASSEMBLER, substituiremos os mnemônicos por seus dígitos hexadecimais e utilizaremos o MONITOR para colocar o programa em forma de bits no computador. A tabela completa das instruções está no apêndice 2; no entanto, cada vez que uma instrução for apresentada, daremos também qual é o seu código hexadecimal.

FIGURA 2.1 - Principais registros internos do Z-80.



Nos MSX os programas em Linguagem de Máquina são tratados como sub-rotinas de um programa em BASIC, as quais são chamadas pela função USR (em BASIC, abreviação de USeR).

Para definir o ponto de entrada da sua sub-rotina em Linguagem de Máquina deve-se usar o comando:

```
DEFUSRY=XXXX
```

onde XXXX é o ponto de entreda e Y pode variar de 0 a 9, ou seja, você pode definir até 10 pontos de entrada para sub-rotinas em Linguagem de Máquina em um programa BASIC.

Para passar o controle do micro para essas sub-rotinas pode ser usada uma dessas duas formas:

```
PRINT USRY (número ou variável)
```

ou

```
váriavel = USRY (número ou variável)
```

nas quais Y pode variar de 0 a 9.

Essa sintaxe permite que troquemos valores de uma variável de um programa BASIC com uma sub-rotina em linguagem de máquina ou que esses valores sejam impressos no vídeo. Veremos como se processa essa troca mais para o fim deste capítulo.

Se o programa em linguagem de máquina, é tratado pelo BASIC como uma sub-rotina é de se esperar que exista uma instrução similar ao RETURN do BASIC em Linguagem de Máquina.

O código dessa instrução é C9H e seu mnemônico correspondente é RET (RETurn).

A instrução RET ao final de programa é muito importante pois, ao contrário de um programa BASIC, a execução de um programa em Linguagem de Máquina não pára na última instrução. O microprocessador tentará continuar a executar as instruções que se seguem na memória e coisas muito estranhas poderão acontecer.

O mesmo poderá ocorrer se houver erro de lógica no programa e você notará que nem as teclas CONTROL/STOP poderão lhe devolver o controle do micro.

A única maneira de fazer as coisas voltarem ao normal será desligar e religar o computador, ou causar um RESET total na máquina.

Eventualmente você poderá usar um pequeno "truque". Antes de executar o programa em Linguagem de Máquina, comande:

```
POKE &HFBB0,255
```

Assim, sempre (ou quase) que você pressionar simultaneamente as teclas CTRL+SHIFT+LGRA+RGRA no Expert ou CTRL+SHIFT+CODE+GRAPH no Hotbit, o micro voltará a operar em BASIC.

A INSTRUÇÃO LD

A instrução LD permite colocar números dentro dos registros internos do microprocessador (obviamente estes números poderão ser de 00H a FFH

no máximo para cada registro), copiar os dados de um registro em outro ou em algum endereço da memória e vice-versa; assim, existirão várias instruções LD, cada uma com seu prórpio código hexadeciaml. (LD é abreviação de LoaD = carregue).

Vamos analisar inicialmente como colocar números diretamente nos vários registros.

A instrução, de modo genérico, possui a seguinte estrutura:

```
LD registro, dado
```

Por exemplo:

```
LD E, 2AH
```

(que equivale a dizer LD E, 42), colocará o número 2AH (42) no registro E. Esta instrução corresponde a 2 bytes em código hexadecimal, um para o código de operação (no caso, carregue o registro E, LD E,) e o outro para o dado (número) que será colocado. Assim, para os vários registros, teremos os seguintes códigos de operação e códigos hexadecimais:

FIGURA 2.2 - A instrução "LD" registro,dado (8 bits).

INSTRUÇÃO	CÓDIGO
LD A, dado	3EH + 1 byte para o dado
LD B, dado	06H + 1 byte para o dado
LD C, dado	0EH + 1 byte para o dado
LD D, dado	16H + 1 byte para o dado
LD E, dado	1EH + 1 byte para o dado
LD H, dado	26H + 1 byte para o dado
LD L, dado	2EH + 1 byte para o dado

Faremos então um programa que carrega o registro B com 00H e o registro C com 2AH:

FIGURA 2.3 - Programa exemplo.

```
C000 0600      LD    B,00H        ;carrega B com 0
C002 0E2A      LD    C,2AH        ;carrega C com 2AH
C004 C9        RET                ;retorna ao BASIC
```

Podemos então colocá-lo, por exemplo, a partir do endereço C000H (lembre-se de reservá-lo), usando cinco vezes o POKE, ou então, usando o programa MONITOR para introduzir as três instruções.

Com o programa monitor, basta digitar:

```
>M &HC000    (e RETURN)

C000   00   06
C001   00   00
C002   00   0E
C003   00   2A
C004   00   C9
C005   00
```

e teclar ESC para sair do modo M.

Se você quiser verificar se o programa foi introduzido corretamente, um dump pode ser feito. Mas se você digitar

```
L &HC000
```

Você terá, além dos conteúdos dos bytes a partir do endereço C000H, o código mnemônico correspondente à cada grupo de bytes que formam uma instrução. O processo de decodificação dos bytes na memória em mnemônicos da Linguagem Assembly é denominado "disassembler".

Ao usar o comando L você deve ter obtido a tela da figura 2.4 .

FIGURA 2.4 - Programa Disassemblado.

```
>L &HC000
%-16384  LD B,0          06 00
%-16382  LD C,42         0E 2A
%-16380  RET             C9
%-16379  DI              F3
%-16378  LD HL,62460     21 FC F3
%-16375  JR 16370        18 03
%-16373  LD HL,62465     21 01 F4
%-16370  CALL 162        CD A2 00
%-16367  LD DE,62470     11 06 F4
%-16364  LD BC,5         01 05 00
%-16361  LDIR            ED B0
%-16359  RET             C9
%-16358  LD A,(HL)       7E
%-16357  AND A           A7
%-16356  RET Z           C8
%-16355  CALL 162        CD A2 00
%-16352  INC HL          23
%-16351  JR 16358        18 F7
%-16349  LD HL,29000     21 50 71
%-16346  CALL 28698      CD 1A 70
```

PROGRAMA DIGITADO

RESTANTE DA MEMÓRIA
(LIXO)

NÃO NECESSARIAMENTE
UM PROGRAMA

Saindo fora do monitor comande:

DEF USR=&HC000 e A=USR (0)

A função DEFUSR definiu o ponto de entrada no programa e a função USR executou o programa em Linguagem de Máquina.

Até agora pode parecer meio inútil e sem sentido a Linguagem de Máquina, mas após aprendermos algumas outras instruções poderemos fazer algo mais prático.

No programa anterior nós carregamos o registro B com um valor e o C com outro. Porém, os registros B e C podem ser juntados, formando um par de 16 bits.

Por isso há também a possibilidade de carregar diretamente pares de registros. Neste caso, a instrução possui 3 bytes, 1 para o código da instrução e 2 para o número que terá 16 bits. Note que o byte menos significativo do número de 16 bits vem antes do byte mais significativo.

Assim, temos na figura 2.5 as instruções:

FIGURA 2.5 - A instrução LD para pares de registros

INSTRUÇÃO	CÓDIGO
LD BC,dado	01H + 2 bytes para o dado
LD DE,dado	11H + 2 bytes para o dado
LD HL,dado	21H + 2 bytes para o dado

O programa da figura 2.2 poderia ser simplificado ficando como mostra a figura 2.6.

FIGURA 2.6 - Programa que carrega o par BC com 43.

```
C000 012B00    LD    BC,002BH     ;carrega BC com 2BH
C003 C9        RET                ;retorna ao BASIC
```

Note a inversão: 2BH é colocado antes de 00H na memória !

A TRANSFERÊNCIA ENTRE REGISTROS

Veremos agora como passar números de um registro para o outro. Temos todas as combinações

possíveis da seguinte instrução genérica:

LD registro,registro

Essa instrução copia o conteúdo do registro da direita no registro da esquerda, não alterando o conteúdo do registro da direita; essa instrução equivale a apenas um byte pois não exige a especificação de nenhum dado. Para facilitar contruiremos uma tabela que possui todas as combinações possíveis desta instrução.

LD	A	B	C	D	E	H	L
A	7FH	78H	79H	7AH	7BH	7CH	7DH
B	47H	40H	41H	42H	43H	44H	45H
C	4FH	48H	49H	4AH	4BH	4CH	4DH
D	57H	50H	51H	52H	53H	54H	55H
E	5FH	58H	59H	5AH	5BH	5CH	5DH
H	67H	60H	61H	62H	63H	64H	65H
L	6FH	68H	69H	6AH	6BH	6CH	6DH

Para utilizá-la, use sempre primeiro a coluna da vertical à esquerda e, a seguir, a linha horizontal superior; por exemplo:

LD C,D equivale a 4AH

(copie o conteúdo do registro D, sem alterá-lo, no registro C)

LD E,E equivale a 5BH

(copie o conteúdo do registro E no registro E). Como você pode pode notar, essa instrucão não faz absolutamente NADA!).

Experimente agora o seguinte programa, que copia o conteúdo dos registros H e E, previamente carregados, nos registros B e C, respectivamente.

FIGURA 2.7 - Programa para copiar registros.

```
C000 2600      LD    H,00H        ;carrega H com 0
C002 1E2A      LD    E,2AH        ;carrega E com 2AH
C004 44        LD    B,H          ;copia H em B
C005 4B        LD    C,E          ;copia E em C
C006 C9        RET                ;volta ao BASIC
```

Utilize então o monitor para colocar os códigos do programa na memória.

Perceba que este programa "carrega" o registro H com 00H e o registro E com 2AH (42) e, a seguir, copia o registro H em B e o registro E em C, fazendo BC=002AH. Desse modo, você obterá novamente 42.

OBSERVAÇÃO: Não existe instrução para copiar de uma só vez números de um par de registros para outro par de registros. Por exemplo, a instrução LD BC,HL não é válida, e deve ser substituída por:

```
LD B,H
```

e

```
LD C,L
```

Além disso, não é possível tentar copiar o conteúdo de um registro para um par ou vice-versa; portanto, instruções do tipo

```
LD BC,A
LD D,HL
```

NÃO existem !!!

A título de esclarecimento, podemos agora fazer uma analogia com o BASIC. As intruões LD registro,dado de 1 byte ou LD par de registros,dado de 2 bytes podem ser associadas à instrução LET variável=número; por exemplo, LD B,10 com LET X=16. A instrução LD registro,registro pode ser associada à LET variável= variável; por exemplo LD H,L com LET X=Y.

ENDEREÇAMENTO DIRETO E INDIRETO

Até aqui, aprendemos como colocar números diretamente nos registros internos do microprocessador e como copiar dados de um registro em outro. Veremos agora como copiar o conteúdo dos registros internos

nos registros da memória, e vice-versa. Veja como isto amplia nosso universo: até agora só podíamos colocar dados em sete registros mas, com essas novas instruções, teremos a disposição toda memória RAM. Naturalmente, devemos tomar cuidado para não interferirmos com os demais programas, por enquanto este problema não nos atinge pois estamos utilizando uma região de memória reservada no fim da RAM.

Se você escreve, em BASIC:

```
LET X=PEEK (&HC000)
```

O que isto significa ?

Ora, a variável X irá assumir um valor entre 0 e 255, que corresponderá ao que está no endereço C000H .

Em Linguagem de Máquina existe uma instrução análoga, mas que funciona apenas com o registro A. Sua forma genérica é:

```
LD A,(endereço)
```

cujo código é 3AH+2 bytes para o endereço.

Note que os parênteses indicam a seguinte idéia: copie no registro A o conteúdo do endereço.

De fato, a instrução LD A, endereço (sem parênteses) não é válida porque não podemos colocar um número maior que 255 dentro de um único registro.

Esse tipo de instrução é chamado endereçamento direto, pois podemos dizer diretamente na instrução em que o endereço da memória está o que queremos colocar no registro A.

Observe que esta instrução equivale a 3 bytes: um para o código de operação (no caso 3AH) e dois para dizer qual endereço cujo conteúdo devemos copiar no registro A (lembre-se da inversão: LSB MSB).

Assim como o PEEK tem sua função inversa (o POKE) a instrução que acabamos de ver também tem sua função inversa:

```
LD (endereço) , A
```

Essa instrução também equivale a 3 bytes, pois precisamos fornecer ao microprocessador o endereço no qual será inserido o valor do registro A.

Essas duas últimas instruções ampliam bastante o nosso universo, pois agora podemos obter o conteúdo de qualquer byte da memória ou inserir nela

qualquer valor (no caso de memória RAM).

Vamos apresentar, na figura 2.8, um programa que troca o conteúdo de dois bytes da memória.

FIGURA 2.8 - Troca de dados da memória.

```
C000 3A0EC0   LD    A,(0C00EH)  ;insere 1º
                                  byte em A
C003 47       LD    B,A         ;insere A em B
C004 3A0FC0   LD    A,(0C00FH)  ;insere 2º
                                  byte em A
C007 320EC0   LD    (0C00EH),A  ;insere A no 1º byte
C00A 78       LD    A,B         ;carrega A com B
C00B 320FC0   LD    (0C00FH),A  ;insere A
                                  no 2º byte
C00E C9       RET               ;retorna ao BASIC
C00F FF       DB    0FFH        ;1º valor
C010 3A       DB    3AH         ;2º valor
```

Ao ser executado, o programa troca o valor dos bytes C00EH e C00FH.

Primeiramente o valor do primeiro byte é carregado no registro A. Em seguida, esse valor é armazenado temporariamente no registro B, assim podemos carregar o segundo byte no acumulador, e "pokeá-lo" no primeiro byte. Após essa operação, o antigo valor do primeiro byte é recuperado do registro B é inserido no segundo byte.

OBSERVAÇÃO: Colocamos os números após o endereço C00EH por ser o primeiro byte livre após o programa e, como explicado anteriormente, é nessa região que colocaremos as variáveis que não cabem nos registros. Cuidado para não colocar números diretamente em bytes ocupados pelo programa, pois você estará alterando as instruções.

Vimos então que, apesar de só podermos usar uma instrução análoga ao PEEK com o registro A, isto é contornável pois podemos copiar os dados de um registro em outro, como no exemplo anterior. Note no entanto que nós fizemos, como exemplo, uma instrução do tipo PEEK número; e se fizéssemos PEEK variável, por exemplo, LET X = PEEK (Y) (onde Y pode valer de 0 a 65535)? De fato, em Linguagem de Máquina é possível a instrução do tipo

LD registro,(par de registro)

que significa "copie no registro indicado (à esquerda) o conteúdo da memória cujo endereço é dado pelo par

de registros entre parênteses (à direita). No entanto, nem todas as combinações são possíveis.Veja a fig.2.9.

FIGURA 2.9 - A instrução LD registro, (par de registros)

INSTRUÇÃO	CÓDIGO
LD A, (BC)	0AH
LD A, (DE)	1AH
LD A, (HL)	7EH
LD B, (HL)	46H
LD C, (HL)	4EH
LD D, (HL)	56H
LD E, (HL)	5EH
LD H, (HL)	66H
LD L, (HL)	6EH

Este tipo de instrução é chamado endereçamento indireto por pares de registro pois o endereço daquilo que queremos colocar nos registros é dado indiretamente através dos pares BC, DE e HL, para o registro A, e somente do par HL para os demais registros (B, C, D, H e L).

Podemos perceber que o registro A parece ser privilegiado em relação aos demais; de fato, ele tem várias propriedades que os outros não têm, e inclusive um nome especial: Acumulador (no próximo capítulo veremos com mais detalhes suas propriedades especiais).

Assim como existem instruções de endereçamento indireto para se obter o valor de um byte, existem as instruções de endereçamento indireto para inserir valores na memória. Veja os seus mnemônicos e códigos na figura 2.10.

FIGURA 2.10 - Transferência registros => memória.

INSTRUÇÃO	CÓDIGO
LD (NNMM),A	32H MMNN
LD (BC),A	02H
LD (DE),A	12H
LD (HL),A	77H
LD (HL),B	70H

INSTRUÇÃO	CÓDIGO
LD (HL),C	71H
LD (HL),D	72H
LD (HL),E	73H
LD (HL),H	74H
LD (HL),L	75H

Perceba novamente como o acumulador é privilegiado; ele é o único que permite endereçamento direto, colocando o número desejado da memória e, ao contrário dos demais, endereçamento indireto não só através do par HL mas também dos pares BC e DE.

Vamos montar o programa para a troca de 2 bytes usando endereçamento indireto (figura 2.11).

FIGURA 2.11 - Programa de troca de dois bytes.

```
C000 010CC0    LD    BC,0C00CH   ;insere 1º endereço
                                   em BC
C003 210DC0    LD    HL,0C00DH   ;insere 2º endereço
                                   em HL
C006 0A        LD    A,(BC)      ;insere 1º byte em A
C007 56        LD    D,(HL)      ;insere 2º byte em D
C008 77        LD    (HL),A      ;insere 1º byte no
                                   2º endereço
C009 7A        LD    A,D         ;transfere D para A
C00A 02        LD    (BC),A      ;insere 2º byte no
                                   1º endereço
C00B C9        RET               ;retorna ao BASIC
```

Perceba que nos programas estamos colocando alguns comentários para facilitar a compreensão (sempre os colocaremos após um ponto e vírgula (;)).

A figura 2.12 mostrará passo a passo os valores dos registros durante a execução do programa.

FIGURA 2.12 - Execução passo a passo de um programa.

LINHA	C00C	C00D	A	B	C	D	E	H	L
1º →	FF	3A		C0	0C				
2º →	FF	3A		C0	0C			C0	0D
3º →	FF	3A	FF	C0	0C			C0	0D
4º →	FF	3A	FF	C0	0C	3A		C0	0D
5º →	FF	FF	FF	C0	0C	3A		C0	0D
6º →	FF	FF	3A	C0	0C	3A		C0	0D
7º →	3A	FF	3A	C0	0C	3A		C0	0D

Muito cuidado com as instruções do tipo LD (endereço), A. Suponhamos que você tenha

```
LD (4023),A
```

seu primeiro byte deve ser 32H, que é seu código

de operação seguido do endereço invertido, ou seja, 2340H. Chamamos a atenção para o fato de que a tendência natural é escrever esta instrução ao contrário, ou seja, o endereço antes e o código depois. Para qualquer instrução em Linguagem de Máquina é obrigatório sempre se colocar antes o código de operação. Senão, vejamos: se colocarmos, por exemplo, 324A7CH, essa sequência será interpretada pelo microprocessador como LD (7C4AH),A pois o número 32H, quando interpretado como instrução, ordena que o microprocessador copie o conteúdo do acumulador na memória indicada pelos próximos 2 bytes (sempre levando em conta a inversão).

Ao escrever 4AH, 7CH e 32H, o microprocessador interpretará o seguinte:

```
4AH LD C,D
7CH LD A,H
32H LD (????),A
```

onde o endereço (????H) será interpretado como sendo os próximos 2 bytes da memória!

Lembre-se sempre disso para evitar as já famosas "coisas estranhas" mencionadas anteriormente. Perceba que o fato de um determinado byte significar instrução ou dado depende apenas da sua posição no programa. Por exemplo, acabamos de ver que 4AH se interpretado como instrução significa LD C,D ; mas se quisermos fazer LD (7C4AH),A, o byte 4AH será parte dos dados de instrução.

Completaremos agora a lista de instruções LD usando endereçamento direto para pares de registros. O que você acha que significa a instrução LD HL,(FC4AH). Como é possível copiar o conteúdo do endereço FC4AH, que tem 1 byte apenas, no par de registro HL, que tem 2 bytes? De fato, essa instrução funciona da seguinte maneira: copia no registro L (o menos significativo) o conteúdo da memória FC4AH e no registro H (o mais significativo) o conteúdo da memória seguinte, FC4BH. Isso em BASIC poderia ser associado a

```
LET X = PEEK (&HFC4A) + 256 * PEEK (&HFC4B)
```

Você se lembra qual a variável contida nesse endereço ?

Note bem a diferença entre as seguintes instruções:

```
LD HL,0FC4AH
LD HL,(0FC4AH)
```

A segunda foi a que acabamos de descrever; a primeira foi descrita anteriormente e significa: carregue o par HL com o número FC4AH e L = 04H , ou seja, H = FCH e L = 4AH.

Naturalmente, para a segunda instrução, existe também a sua inversa:

```
LD (0FC4AH),HL
```

que copia o registro L no endereço FC4AH e o H no endereço FC4BH o que em BASIC equivaleria a

```
POKE &HFC4A, X MOD 256

POKE &HFC4B, X / 256
```

Você saberia explicar porque?
Assim temos as seguintes instruções:

INSTRUÇÃO	CÓDIGO
LD BC,(endereço)	ED4BH + (2 bytes para o endereço)
LD DE,(endereço)	ED5BH + (2 bytes para o endereço)
LD HL,(endereço)	2AH + (2 bytes para o endereço)
LD (endereço),BC	ED43H + (2 bytes para o endereço)
LD (endereço),DE	ED53H + (2 bytes para o endereço)
LD (endereço),HL	22H + (2 bytes para o endereço)

Note que aqui aparecem algumas instruções cujo código de operação tem 2 bytes. Para todas elas, o primeiro byte corresponde a EDH. Isso porque, se todos os códigos de operação tivessem apenas 1 byte, teríamos somente 256 códigos diferentes. Portanto, para ampliar o número de instruções, algumas têm o mesmo código que as outras, precedido por EDH e outras, ainda, precedido por CBH . Por exemplo, o código 4BH , sozinho, corresponde à instrução (LD C,E) e, precedido por EDH , ou seja, ED4BH, corresponde a LD BC, (endereço).

OBSERVAÇÃO: Não existe endereçamento indireto para pares de registros.

Para finalizar, apresentaremos a instrução cujo código de operação é 36H, a saber:

```
LD (HL),número (36H+1 byte para o número)
```

que permite colocar um número de 1 byte diretamente na memória cujo endereço é dado pelo par HL (apenas o par HL permite essa instrução).

Note que não existe instrução para copiar diretamente um registro de memória em outro, assim como fazemos nos registros internos. Como você faria então para copiar, por exemplo, o conteúdo do endereço C010H no endereço C020H.

É necessário que o dado seja antes copiado num registro interno do microprocessador, de preferência o acumulador. (Por quê?) Bastaria fazer o apresentado na figura 2.13 .

FIGURA 2.13 - Transferência de um byte na memória.

```
C000 3A10C0   LD    A,(0C010H)
C003 3220C0   LD    (0C020H),A
C006 C9       RET
```

Teste o programa fazendo o endereço C010H conter FAH e verificando o conteúdo do byte C020H antes e depois de executado o programa.

Anteriormente dissemos que o único registro que permitia um LD direto de um endereço da memória era o acumulador:

```
LD A,(endereço)
```

Podemos, com um pequeno "truque", fazer o mesmo para os registros C, E ou L. Observe um exemplo para o registro E onde simularemos a instrução LD E,(endereço) que, de fato, não existe:

```
C000 ED5B07C0  LD    DE,(0C007H) ;copia em E o
                                  conteúdo de C007H
C004 1600      LD    D,0         ;zera o registro D
C006 C9        RET               ;retorna ao BASIC
C007 0000   FA: DEFB  0           ;variável
```

OBSERVAÇÃO: Lembre-se sempre que estamos usando, para colocar as variáveis desses programas em Linguagem de Máquina, a região de memória após o fim dos mesmos endereços. Por exemplo, o último programa tinha 6 bytes e, portanto, começava no endereço C000H, terminando no endereço C005H; assim colocamos a nossa variável no endereço C007H.

Tome muito cuidado para não errar nesse procedimento, pois, caso contrário, você poderá modificar o próprio programa ou fazer um programa que use instruções dele mesmo como variáveis, e novamente irão aparecer as "coisas estranhas".

Vimos até agora que pela instrução LD (LoaD) podemos carregar os registros com valores da memória, transferí-los de um registro para outro ou devolvê-los para a memória.

Porém os MSX, devido ao seu processador de vídeo, possui um banco de memória RAM exclusivo para este. Ou seja, o Z-80 não pode acessar diretamente esses bytes. O Z-80 tem que se comunicar com o processador de vídeo para obter o valor de um byte na VRAM ou para escrever em um byte da mesma.

O processo para se obter o valor de um byte "via" processador de vídeo não é muito simples (pelo menos por enquanto). Pensando nisso, o BIOS do MSX já possui duas rotinas para se poder ler ou gravar um byte na VRAM. São elas:

WRTVRM	(WRite To VRaM)
RDVRM	(ReaD from VRaM)

Antes de chamar a rotina WRTVRM deve ser carregado no par HL o endereço da VRAM no qual vai ser gravado o byte. O valor do byte a ser gravado deverá estar no registrador A.

O funcionamento da rotina RDVRM é muito semelhante. O par HL deverá conter o endereço do byte a ser lido e esse valor retornará no registro A.

Note que o dado a ser gravado ou o dado lido estarão obrigatoriamente no registro A. Não existem rotinas para se ler ou gravar dados na VRAM com os outros registros.

As rotinas serão chamadas com o comando CALL (não confunda com o comando CALL do BASIC) que é um equivalente ao GOSUB do BASIC. Ou seja, quando essa instrução é executada, a execução vai para um endereço indicado e executa as instruções até encontrar uma instrução do tipo RET.

A sintaxe do comando CALL é a seguinte:

CALL (Endereço) CDH+2 bytes para o endereço.

É importante ressaltar que o valor de nenhum dos outros registros é afetado após a chamada dessas rotinas.

Observe o exemplo da figura 2.12.

Aparentemente ele não tem muita utilidade.

Afinal ele só transfere o byte em C00AH para o endereço 6912 da VRAM. Mas se fizermos o programinha em BASIC da figura 2.14 poderemos ter alguma coisa útil.

FIGURA 2.14 - Programa que grava um byte na VRAM.

```
C000 21001B   LD    HL,6912      ;carrega HL com
                                   endereço da VRAM
C003 3A0AC0   LD    A,(0C00AH)   ;carrega A com o
                                   conteúdo de 0C00H
                                   da RAM
C006 CD0000   CALL  WRTVRM       ;grava o byte
                                   na VRAM
C009 C9       RET                ;retorna ao BASIC
```

FIGURA 2.15 - Programa auxiliar em BASIC.

```
10 SCREEN 1
20 DEFUSR=&HC000
30 SPRITE$(1)=CHR$(255)+ STRING$(6,
   &B10000001)+CHR$(255)
40 PUTSPRITE0,(128,96),8,1
50 FOR L=0 TO 255
60 POKE &HC00A,L
70 A=USR(0)
80 NEXT
90 GOTO 50
100 END
```

Ao ser executado, o programa em BASIC irá definir um sprite e coloca-lo-á no meio da tela.

Em seguida é feito um loop (a partir da linha 50) que "pokeia" no endereço C00AH o valor da variável L (0-255) e chama o programa em Linguagem de Máquina da figura 2.12 .

E simplesmente o que aconte? O sprite começa a se mover verticalmente pela tela.

O segredo desse movimento está na alteração do byte 6912 da VRAM. É nesse byte que é guardada a coordenada Y do sprite da camada 0.

TROCANDO VARIÁVEIS COM O BASIC

Com as instruções vistas até agora, temos um vocabulário em Linguagem de Máquina que nos permite gravar e recuperar dados da memória. Com isso podemos trocar valores de variáveis com o BASIC.

Você já deve ter reparado que quando é executado um comando do tipo:

PRINT USR(0) ou X=USR(A)

o número, string ou variável de parâmetro da função USR é impresso na tela do vídeo ou atribuído à variável.

Isso ocorre porque o BASIC MSX permite a troca automática de valores entre um programa BASIC e um programa em Linguagem de Máquina.

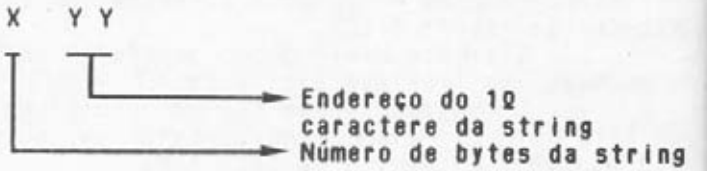
O valor das variáveis fica armazenado a partir do endereço F7F6H obedecendo a certos parâmetros para cada tipo e variável:

Variáveis inteiras - os endereços F7F8H e F7F9H contêm na forma LSB e MSB o valor da variável.

Variáveis de simples e dupla precisão - os valores dessas variáveis ficam armazenados a partir do endereço F7F6H, ocupando 4 bytes para as variáveis de simples precisão e 8 para as de dupla precisão.

Variáveis string - o endereço F7F8H apontará para o descritor da string. O descritor da string é composto de 3 bytes como mostra a figura 2.14 .

FIGURA 2.16 - Formato do descritor da string.



Para que o programa em Linguagem de Máquina possa detectar o tipo da variável que serve de parâmetro para a função USR, existe a variável de sistema VALTYP (F663H) que contém:

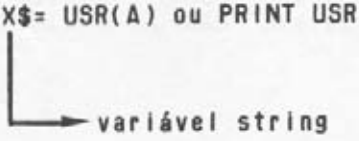
2 - variáveis inteiras.
4 - variáveis em simples precisão
8 - variáveis em dupla precisão
3 - variáveis tipo string

Quando o controle do micro é devolvido ao Interpretador BASIC, os valores a partir do byte F7F8H são atribuídos variável ou impressos na tela conforme o comando pelo qual a USR foi chamada.

A variável de sistema VALTYP também deve conter o tipo do valor de retorno. Isso evita que o interpretador tente atribuir a variáveis de um certo tipo (string, valores inteiros, etc.) um valor de retorno de um tipo diferente.

Por exemplo, imaginemos que foi feito um programa em Linguagem de Máquina que retorne uma variável tipo string. Ao terminar a execução, a variável de sistema VALTYP (F663H) deve conter o valor 3.

Isso indicará ao Interpretador o que fazer com os bytes a partir de F7F8H e a só aceitar linhas com a seguinte sintaxe:



Vejamos dois exemplos de como trocar variáveis com o BASIC.

Existem muitas variáveis de sistema que apontam para certos endereços de memória. E para tal são armazenadas na forma LSB e MSB. Para conseguir esses endereços em BASIC usamos fórmula:

```
PRINT PEEK(endereço)+256*PEEK(endereço+1)
```

O programa a seguir (figura 2.15) fará exatamente isso, jogando o valor em uma variável inteira. Será dado como parâmetro o endereço a ser consultado (também em uma variável inteira).

Você deverá notar no programa uma instrução

que ainda não conhecemos que é o

INC HL (código 23H)

que nada mais faz do que INCrementar em uma unidade o valor contido no par HL.

FIGURA 2.17 - Programa de consulta de variáveis.

```
C000 2AF8F7    LD    HL,(0F7F8H) ;aponta HL para
                                   a variável
C003 7E        LD    A,(HL)      ;carrega em A o 1º
                                   byte da variável
C004 32F8F7    LD    (0F7F8H),A  ;armazena em F7F8H
C007 23        INC   HL          ;soma 1 em HL
C008 7E        LD    A,(HL)      ;carrega em A o 2º
                                   byte da variável
C009 000000    LD    (0F7F9),A   ;armazena em F7F9
C00C C9        RET               ;retorna ao BASIC
```

Vamos testar esse programa com a variável de sistema HIMEM digitando:

DEFUSR=&HC000 (RETURN)

PRINT HEX$(USR(&HFC4A))

Você deverá obter como resultado o endereço de limite para um programa BASIC e suas respectivas variáveis.

3

INSTRUÇÕES ARITMÉTICAS

Provavelmente após ter aprendido o conceito de registro e as várias combinações da instrução LD, você deve ter pensado se seria possível efetuar cálculos com os registros, assim como fazemos em BASIC com as variáveis de um programa.

A resposta para essa pergunta não chega a ser muito animadora pois a Linguagem de Máquina permite fazer diretamente apenas soma e subtração e, como você já deve ter notado, lida apenas com números inteiros.

AS INSTRUÇÕES DE ADIÇÃO

Vamos iniciar com as instruções de adição de registros e de pares de registros:

FIGURA 3.1 - A instrução ADD entre registros.

INSTRUÇÕES	CÓDIGO
ADD A,A	87H
ADD A,B	80H
ADD A,C	81H
ADD A,D	82H
ADD A,E	83H
ADD A,H	84H
ADD A,L	85H

INSTRUÇÕES	CÓDIGO
ADD HL,BC	09H
ADD HL,DE	19H
ADD HL,HL	29H

Essas instruções significam: some o conteúdo do registro ou par de registros da direita para a esquerda, e deixe o resultado no registro ou par de registros da esquerda. Os registros da direita, portanto, não são alterados. Em BASIC, uma analogia poderia ser feita com

```
LET X=X+Y
```

Novamente notamos o registro A, ou seja, o acumulador, como sendo privilegiado; de fato, ele é o alvo de todas as operações aritméticas entre registros ou entre registros e memória, permanecendo nele o resultado da operação. No caso de par de registros, o par privilegiado é o HL. Não existem, portanto, instruções do tipo ADD D,E ou ADD BC,DE. Qualquer operação aritmética deve utilizar o acumulador A ou o par HL.

Vamos então somar o conteúdo do registro D com o registro E, utilizando uma sub-rotina em Linguagem de Máquina:

```
C000 1E22     LD    E,34        ;carrega E com 34
C002 1642     LD    D,66        ;carrega D com 66
C004 7A       LD    A,D         ;carrega A com con-
                                  teúdo de D
C005 83       ADD   A,E         ;soma o conteúdo de
                                  E em A
C006 21F6F7   LD    HL,0F7F6H   ;aponta HL para a
                                  variável de troca
                                  com o BASIC
C009 3600     LD    (HL),0      ;zera o 1º byte
                                  (MSB)
C00B 23       INC   HL          ;soma 1 em HL
C00C 77       LD    (HL),A      ;coloca o conteúdo
                                  de A no byte
C00D 3E02     LD    A,2         ;carrega A com 2
C00F 2163FF   LD    HL,0FF63H   ;aponta HL para
                                  FF63H
C012 77       LD    (HL),A      ;insere 2 no ende-
                                  reço FF63 para
                                  indicar ao BASIC
                                  valor inteiro
C013 C9       RET               ;retorna ao BASIC
```

Note que a soma não pode ser feita diretamente. Devemos obrigatoriamente, usar o acumulador. Coloque o programa na memória e execute-o (usando o monitor). Você deverá obter o número 100 (=34 + 66). Provavelmente lhe deve ter surgido a seguinte pergunta: e se a soma for maior que 255 (ou seja FFH) que é o máximo que "cabe" num registro? E, no caso de pares de registros, se o resultado for maior que 65535 (FFFFH)?

De fato, em BASIC, quando uma variável "estoura" o limite máximo do computador, o programa pára e aparece uma mensagem de erro.

Para ver o que acontece em Linguagem de Máquina, façamos um exemplo. Carregue o registro BC com o valor máximo e some 1:

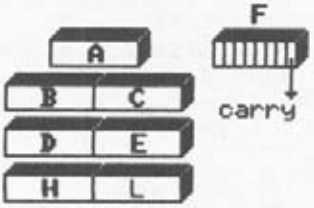
```
C000 210100    LD    HL,1        ;insere 1 em HL
C003 01FF0F    LD    BC,0FFFH    ;carrega BC com o
                                   máximo valor
C006 09        ADD   HL,BC       ;adiciona BC em HL
C007 7C        LD    A,H         ;insere parte alta
                                   de HL em A
C008 32F6F7    LD    (0F7F6H),A  ;insere parte alta
                                   em F7F6H
C00B 7D        LD    A,L         ;insere a parte
                                   baixa em A
C00C 32F7F7    LD    (0F7F7H),A  ;insere a parte
                                   baixa em F7F7H
C00F 3E02      LD    A,2         ;carrega A com 2
C011 3263F6    LD    (0F663H),A  ;insere 2 em F663H
                                   para indicar valor
                                    inteiro ao BASIC
C014 C9        RET               ;retorna ao BASIC
```

Você obtém como resultado o número 0. De fato, pensando em hexadecimal, ao somar 1H ao número FFFFH deveríamos obter 10000H ; mas este último dígito não "cabe" nos registros. Quando isto ocorre, dizemos que houve um "vai um" ou CARRY, assim como ocorre ao somarmos, por exemplo:

```
   1
  19
+ 18    (9 + 8 = 17, "vai um"...)
----
  37
```

O microprocessador assinala este acontecimento em um bit chamado CARRY, que faz parte de um outro registro interno, chamado F, que não pode ser usado diretamente. Este registro armazena bits para várias informações (usualmente estes bits são chamados flags). Assim, sempre que executamos uma instrução de ADD, o bit de CARRY é calculado: se houve "vai um" ele resulta 1, caso contrário, 0.

FIGURA 3.2 - O registro interno F e a flag de CARRY.



Podemos usar o valor desse bit para fazer contas com números maiores que 255 ou até mesmo maiores que 65535. Para isso, usaremos a instrução ADC (ADd with Carry), ou seja, some com o CARRY, que simplesmente adiciona ao resultado obtido de uma soma o valor do CARRY. Teremos então as seguintes instruções:

FIGURA 3.3 - A instrução ADC.

INSTRUÇÃO	CÓDIGO
ADC A,A	8FH
ADC A,B	88H
ADC A,C	89H
ADC A,D	8AH
ADC A,E	8BH

INSTRUÇÃO	CÓDIGO
ADC A,H	8CH
ADC A,L	8DH
ADC HL,BC	ED4AH
ADC HL,DE	ED5AH
ADC HL,HL	ED6AH

Vamos então refazer o programa 3.3 para executar uma soma cujo resultado seja maior que 255.

FIGURA 3.4 - Soma de um número maior que 255.

```
C000 1EC8      LD    E,0C8H
C002 163A      LD    D,03AH
C004 7A        LD    A,D          ;transfere o conteú-
                                    do de D para A
C005 83        ADD   A,E          ;adiciona E em A e
                                    gera CARRY
C006 32F7F7    LD    (0F7F7H),A   ;coloca o resultado
                                    na memória (LSB)
                                    para o BASIC
C009 3E00      LD    A,0          ;zera o acumulador
C00B 8F        ADC   A,A          ;transfere o CARRY
                                    para o acumulador
C00C 32F6F7    LD    (0F7F6H),A   ;coloca o resultado
                                    na memória (MSB)
                                    para o BASIC
C00F 3E02      LD    A,2          ;insere 2 em A
C011 3263F6    LD    (0F663H).A   ;coloca 2 em F663H
                                    para informar valor
                                    inteiro ao BASIC
C014 C9        RET
```

Note que ao somar 200 (C8H) com 58 (3AH) obteremos 258, ou seja, A = 2 e CARRY = 1. A seguir, este CARRY é transferido para o acumulador (ADC A,A) e colocado em F7F6H; deste modo teremos F7F6H = 1 e F7F7H = 2, ou seja, 256 * 1 + 2 = 258.

Para exemplificar o raciocínio a ser adotado, execute o seguinte programa, que soma 13189 (3385H) com 31687 (7BC7H):

```
C000 118533    LD    DE,3385H     ;insere 13189 em DE
C003 21C77B    LD    HL,7BC7H     ;insere 31687 em HL
C006 19        ADD   HL,DE        ;soma DE com HL
C007 7C        LD    A,H          ;carrega H em A
C008 32F7F7    LD    (0F7F7H),A   ;insere A em F7F7H
C00B 7D        LD    A,L          ;carrega L em A
C00C 32F6F7    LD    (0F7F6H),A   ;insere A em F7F6H
C00F 3E02      LD    A,2          ;carrega 2 em A
C011 3263F6    LD    (0F663H),A   ;insere 2 em F663H
C014 C9        RET                ;retorna ao BASIC

C015 1651      LD    D,85H        ;note que 256*01 +
                                    133 = 13189
C017 1E85      LD    E,85H        ;coloca 13189 em DE
C019 267B      LD    H,7BH        ;note que 256*123 +
                                    199 = 31687
C01B 2EC7      LD    L,0C7H       ;coloca 31687 em HL
C01D 7D        LD    A,L          ;insere L em A
C01E 83        ADD   A,E          ;Adiciona E em A e
                                    gera CARRY
C01F 6F        LD    L,A          ;insere o resultado
                                    em L
C020 7C        LD    A,H          ;insere H em A
C021 8A        ADC   A,D          ;adiciona D com A e
                                    com CARRY
C022 32F7F7    LD    (0F7F7H),A   ;armazena o resultado
                                    em F7F7H
C025 7D        LD    A,L          ;carrega L em A
C026 32F6F7    LD    (0F7F6H),A   ;armazena em F7F6H
C029 3E02      LD    A,2          ;carrega A com 2
C02B 3263F6    LD    (0F663H),A   ;insere 2 em F663H
                                    para indicar valor
                                    inteiro ao BASIC
C02E C9        RET                ;retorna ao BASIC
```

Como esperado, os dois programas produzem o mesmo resultado, ou seja, 44876 (=13189 +31687). Duas coisas devem ser observadas: inicialmente, que as instruções LD não afetam o valor do CARRY pois, caso contrário, no segundo as instruções de LD L,A e LD A,H entre ADD A,E e ADC A,D afetariam a nossa soma feita por partes; além disso, a primeira soma deve sempre

ser feita com a instrução ADD, pois não conhecemos o valor inicial do CARRY. Nunca se esqueça da inversão, já mencionada várias vezes, para números de 16 bits, mas que é sempre bom ter em mente: compare as duas primeiras linhas do primeiro programa com as quatro primeiras linhas do segundo!

O que aconteceria se no segundo programa a instrução ADC A,D fosse substituída por ADD A,D? Faça isto utilizando POKE:

```
POKE &HC00C,&H82    (82H é o código de
                     ADD A,D).
```

Execute o programa PRINT USR (0). Você obterá 44620, que é um resultado incorreto. Veja porque: os conteúdos dos registros eram, respectivamente:

```
D = 33H          H = 78H
E = 85H          L = C7H

(3385H = 13189)  (78C7H = 31687)
```

OBSERVAÇÃO: Tente fazer as somas que se seguem em hexadeciamal; não se esqueça que "vai um" só ocorre quando os dígitos somados resultarem maiores que FH (=15) e que para achar o que "fica" devemos subtrair 10H (=16) do resultado. Caso você não consiga, transforme os números em decimal, faça a soma e a seguir converta o resultado hexadecimal. Exemplo:

```
    11
     4AH
+    C8H
  ------
    112H
```

Assim, ao efetuar:

```
LD    A,L
ADD   A,E
LD    L,A
```

obtemos: L = C7H + 85H = 4CH (CARRY=1)
e ao efetuar:

```
LD    A,H
ADC   A,D
LD    H,A
```

obtemos: H = 78H + 33H + 1H = AEH + 1H = AFH

E o resultado seria:
H = AF4CH ou seja: 256 * 175 + 76 = 44876

Se fizermos ADD A,D, teremos H = AEH (ou H = 174); ou seja, HL = AE4CH, que corresponde a 256 * 174 + 76 = 44620.

Vamos agora utilizar a instrução ADC para fazer cálculos com números maiores que 65535. Basta seguir o mesmo raciocínio utilizando no segundo programa, ou seja, somar por partes. Vamos supor que quiséssemos somar os seguintes números:

02A5EFH (= 2 * 65.536 + 165 * 256 + 239 = 173.551)
com F4DEH (=244 * 256 + 222 = 62.686)

(Observação: 65.536 = $(256)^2$).

devemos obter:

173.551 + 62.686 = 236.237

Vamos então dividir os números em três partes, colocando o primeiro número nos registros B, D e H e o segundo nos registros C, E e L. O resultado será colocado nos bytes após o fim dos programas.

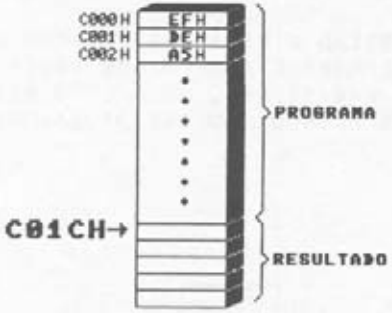
```
C000 06EF     LD   B,0EFH  ─┐
C002 0EDE     LD   C,0DEH  ─┼┐
C004 16A5     LD   D,0A5H  ─┼┼── 1º nº
C006 1EF4     LD   E,0F4H  ─┼┼── 2º nº
C008 2602     LD   H,002H  ─┼┘
C00A 2E00     LD   L,000H  ─┘
C00C 78       LD   A,B
C00D 81       ADD  A,C           ;soma 1º byte
C00E 321CC0   LD   (0C01CH),A    ;insere em C01CH
C011 7A       LD   A,D           ;
C012 88       ADC  A,E           ;soma 2º byte
C013 321DC0   LD   (0C01DH),A    ;insere em C01DH
C016 7C       LD   A,H           ;
C017 8D       ADC  A,L           ;soma 3º byte
C018 321EC0   LD   (0C01EH),A    ;insere em C01EH
C018 C9       RET
```

Novamente não nos interessa o resultado final dos registros BC, que no caso é 61406. O resultado da nossa soma por partes está nos endereços C01CH, C01DH e C01EH que estão após a instrução RET . Portanto, para verificar se o nosso raciocínio está certo, basta fazer:

```
PRINT 65536 * PEEK (&HC01C) + 256 * PEEK
(&HC01D) + PEEK (&HC01E)
```

e obteremos 236.237. Note que a primeira instrução de soma deve ser sempre um ADD, pois não sabemos qual o valor inicial do CARRY e, se por acaso ele for 1, introduziremos um erro no resultado. Outra coisa que é bom observar é o fato de que apenas pares de registros podem ser somados a pares de registros, e que apenas registros podem ser somados a registros; você não pode somar um registro a um par de registros, ou vice-versa. Assim, não existem instruções do tipo ADD HL,A ou ADD A, BC. Perceba que utilizamos os bytes a partir de C01CH para colocar os resultados por estarem após o programa.

FIGURA 3.5 - Visualização da memória.



As instruções de ADD e ADC também podem ser utilizadas para somar constantes numéricas diretamente ao acumulador. Assim, temos:

FIGURA 3.6 - Instruções de soma.

INSTRUÇÃO	CÓDIGO
ADD A,dado	C6H + 1 byte para o dado
ADC A,dado	CEH + 1 byte para o dado

e o resultado ficará no acumulador. Lembre-se que esse registro é privilegiado e todas as operações aritméticas são referidas a ele. Obviamente essas instruções também afetam a flag de CARRY. Entretanto, não podemos somar dados diretamente com pares de registros.

Suponha que você deseje somar um número menor que 256 (por exemplo, 74) com o contido num par de registros (por exemplo, 163 em HL). Isso poderia ser feito como mostra o programa da figura 3.7.

FIGURA 3.7 - Soma de registro com par de registros.

```
C000 210040  LD    HL,4000H     ;carrega HL com 16390
C003 114A00  LD    DE,4AH       ;carrega DE com 74
C006 19      ADD   HL,DE        ;adiciona DE em HL
C007 7A      LD    A,D          ;carrega D em A
C008 32F7F7  LD    (0F7F7H),A   ;carrega F7F7H com A
C00B 7B      LD    A,E          ;carrega E em A
C00C 32F6F7  LD    (0F7F6H),A   ;carrega F7F6 com A
C00F 3E02    LD    A,2          ;carrega A com 2
C011 3263F6  LD    (0F663H),A   ;carrega F663 com A
C014 C9      RET                ;retorna ao BASIC
```

Deveremos obter 16464. Observe que tivemos de utilizar o par de registros DE, colocando 00H no registro D. Entretanto, há ocasiões, que você perceberá à medida que for progredindo, em que não é conveniente desperdiçar registros. Veja que faremos a mesma soma anterior sem usar nenhum outro par de registros:

FIGURA 3.8 - Programa de soma.

```
C000 210040  LD    HL,4000H     ;carrega HL com
                                  16390
C003 7D      LD    A,L          ;insere L no
                                  acumulador
C004 C64A    ADD   A,4AH        ;adiciona 74 em
                                  A e gera CARRY
C006 32F6F7  LD    (0F7F6H),A   ;carrega F7F6H
                                  com A
C009 7C      LD    A,H          ;carrega H em A
C00A CE00    ADC   A,0          ;soma o CARRY
C00C 32F7F7  LD    (0F7F7H),A   ;carrega F7F7
                                  com A
C00F 3E02    LD    A,2          ;carrega A com 2
C011 3263F6  LD    (0F663H),A   ;carrega F663
                                  com A
C014 C9      RET                ;retorna ao BASIC
```

Repare cuidadosamente no uso da instrução ADC A,0, que foi utilizada apenas para somar o valor do CARRY ao registro H. Usando esse mesmo procedimento, tente agora somar um número maior que 255 ao par HL, por exemplo, HL + 266 (você deverá "quebrar" o número em 2 bytes).

A operação de edição é muito utilizada também quando desejamos utilizar uma variável como contador para poder fazer loops de repetição. Mais adiante, aprenderemos como fazer esses loops em Linguagem de Máquina; por enquanto, vamos introduzir a instrução INC registro ou INC par de registros (abreviação de INCrement), que adiciona 1 (um) ao valor do determinado registro (ou par) sem, no entanto, afetar o valor do CARRY. Assim, por exemplo, se um registro contém o valor 255, ao ser incrementado ele terá o valor 0 (zero) e o CARRY permanecerá com o valor que estava anteriormente. Observe que podemos usar esta instrução com todos os registros:

FIGURA 3.9 - INSTRUÇÃO INC.

INSTRUÇÃO	CÓDIGO
INC A	3CH
INC B	04H
INC C	0CH
INC D	14H
INC E	1CH

INSTRUÇÃO	CÓDIGO
INC H	24H
INC L	2CH
INC BC	03H
INC DE	13H
INC HL	23H

Novamente, referindo-se ao BASIC, uma analogia para essa instrução seria:

```
LET X = X + 1
```

Lembre-se sempre da diferença entre ADD e INC, por exemplo, ADD A,1 gerará um valor para o CARRY (0 ou 1) enquanto INC deixará o CARRY inalterado, embora as duas instruções produzam o mesmo resultado no acumulador.

Vamos, então, seguindo os mesmos passos que fizemos com a instrução LD, sair dos registros e ir para a memória. Apenas três instruções são disponíveis, e elas mostram claramente o privilégio do acumulador e do par de registros HL:

FIGURA 3.10 - INSTRUÇÕES COM (HL).

INSTRUÇÃO	CÓDIGO
ADD A,(HL)	86H
ADC A,(HL)	8EH
INC (HL)	34H

que significam, respectivamente: some o conteúdo da memória indicada pelo par HL ao acumulador (sem CARRY e com CARRY) e incremente o conteúdo da memória indicada por HL. Assim, vejamos o procedimento necessário para somar um número (por exemplo, 64) ao conteúdo do endereço C014H (por exemplo, 36):

```
C000 2114C0    LD    HL,0C014H    ;coloca C014H em HL
C003 3E40      LD    A,40H        ;insere 64 em A
C005 86        ADD   A,(HL)       ;soma o conteúdo de
                                    C013H em A
C006 32F7F7    LD    (0F7F7H),A   ;carrega o resultado
                                    em F7F7H
C009 3E00      LD    A,0          ;zera o acumulador
C00B 32F6F7    LD    (0F7F6H),A   ;carrega F7F6H com A
C00E 3E02      LD    A,2          ;insere 2 em A
C010 3263F6    LD    (0F663H),A   ;carrega F663H com 2
C013 C9        RET                ;retorna ao BASIC
```

Comande POKE &HC014, &H36 e execute o programa. Você deverá obter 100, pois irá realizar a conta 64 + 36 = 100. É sempre bom recordar que as variáveis devem ficar após a instrução RET, deixando espaço livre na memória, entre o fim do programa e as variáveis, para eventuais modificações no programa que possam causar aumento do seu tamanho.

Você saberia fazer esse mesmo programa de outra maneira, utilizando endereçamento direto ou indireto e a instrução LD? O que aconteceria se o resultado da soma fosse maior do que 255?

AS INSTRUÇÕES DE SUBTRAÇÃO

Vamos agora à subtração. As instruções são parecidas com as de adição, e também geram CARRY quando o resultado obtido for menor que zero. No entanto, para evitar complicação excessiva, vamos, por enquan-

to, esquecer os números negativos e utilizar o CARRY apenas nas subtrações por partes. Assim, temos:

FIGURA 3.11 - A instrução SUB.

INSTRUÇÃO	CÓDIGO
SUB A,A	97H
SUB A,B	90H
SUB A,C	91H
SUB A,D	92H
SUB A,E	93H
SUB A,H	94H
SUB A,L	95H

que, à semelhança das instruções de adição, subtraem o conteúdo do registro da direita do acumulador, deixando o resultado no acumulador. Embora não exista a instrução SUB para pares de registros, por mais estranho que isso possa parecer, a instrução SBC (subtraia com CARRY - SuBtract with CARRY) existe tanto para registros simples quanto para pares de registros. Esse problema pode ser contornado se conseguirmos garantir que a flag de CARRY esteja em 0 (zero) antes de efetuarmos um SBC para pares de registros; com isto, estaremos simulando um SUB. Eis as instruções:

FIGURA 3.12 - A instrução SBC.

INSTRUÇÃO	CÓDIGO
SBC A,A	9FH
SBC A,B	98H
SBC A,C	99H
SBC A,D	9AH
SBC A,E	9BH

INSTRUÇÃO	CÓDIGO
SBC A,H	9CH
SBC A,L	9DH
SBC HL,BC	ED42H
SBC HL,DE	ED52H
SBC HL,HL	ED62H

Vamos fazer agora uma sub-rotina para zerar o par de registros HL.

Note que a instrução ADD A,0 será utilizada para garantir que a flag de CARRY seja zerada. Tente imaginar o que aconteceria se não colocássemos esta instrução e o CARRY estivesse com o valor 1.

```
C000 C600      ADD   A,0          ;gera CARRY = 0
C002 ED62      SBC   HL,HL        ;subtrai HL e
                                   CARRY de HL
C004 86        ADD   A,(HL)       ;soma o conteúdo
                                   de C013H em A
C005 32F7F7    LD    (0F7F7H),A   ;carrega H em F7F7H
C008 7D        LD    A,L          ;carrega L em A
C009 32F6F7    LD    (0F7F6H),A   ;carrega L em F7F6H
C00C 3E02      LD    A,2          ;insere 2 em A
C00E 3263F6    LD    (0F663H),A   ;carrega F663H com 2
C011 C9        RET                ;retorna ao BASIC
```

Podemos também subtrair constantes numéricas do acumulador:

FIGURA 3.13 - SUB com constantes numéricas.

INSTRUÇÃO	CÓDIGO
SUB A,dado	D6H + 1 byte para o dado
SBC A,dado	DEH + 1 byte para o dado

Iremos, para exemplificar, fazer uma subtração por partes com números de 3 bytes, usando os mesmos números do exemplo da soma, ou seja:

173.551 - 62.686 = 110.865
(02A5EFH) - (F4DEH)

```
C000 06EF      LD    B,0EFH       ;1º nº em H, D e B
C002 0EDE      LD    C,0DEH       ;2º nº em L, E e C
C004 16A5      LD    D,0A5H       ;
C006 1EF4      LD    E,0F4H       ;
C008 2602      LD    H,002H       ;
C00A 2E00      LD    L,000H       ;
C00C 78        LD    A,B          ;
C00D 000000    SUB   A,C          ;subtrai o 1º byte
C010 321DC0    LD    (0C01DH),A   ;coloca o resultado
                                   em C01DH
C013 7A        LD    A,D          ;
C014 99        SBC   A,C          ;subtrai com o CARRY
                                   o 2º byte
C015 321EC0    LD    (0C01EH),A   ;coloca o resultado
                                   em C01EH
C018 7C        LD    A,H          ;
C019 9D        SBC   A,L          ;subtrai com CARRY o
                                   3º byte
C01A 321FC0    LD    (0C01FH),A   ;insere o resultado
                                   em C01FH
C01D C9        RET
```

Experimente colocar você mesmo os códigos de máquina utilizando o apêndice 2 e o exemplo de soma (programa 3.6) para ajudá-lo. A seguir, coloque a sub-rotina na memória. Verifique se o resultado que está está nos 3 bytes a partir de C01DH corresponde ao resultado esperado.

Vale a pena lembrar que não existe instrução para subtrair diretamente um registro de um par e vice-versa. Para isso é necessário usar um procedimento análogo ao utilizado para somar um registro a um dado par. (Experimente fazer um programa, baseado nos exemplos fornecidos anteriormente: subtraia o número 74 do registro HL, previamente carregado com 16.390 e 266 de HL carregado com 16.390. Finalmente, para terminar nossa analogia com adição, introduziremos as instruções de DEC (abreviação de DECrement), e a subtração utilizando o conteúdo da memória.

FIGURA 3.14 - A instrução DEC.

INSTRUÇÃO	CÓDIGO
DEC A	3DH
DEC B	05H
DEC C	0DH
DEC D	15H
DEC E	1DH
DEC H	25H
DEC L	2DH

INSTRUÇÃO	CÓDIGO
DEC BC	0BH
DEC DE	1BH
DEC HL	2BH
DEC (HL)	35H
SUB A,(HL)	96H
SBC A,(HL)	9EH

A instrução DEC subtrai 1 (um) do valor de dado registro (ou par) ou sem afetar o CARRY.

Você saberia explicar a diferença entre as instruções DEC HL e DEC (HL) (ou entre INC HL e INC (HL))?

Para recordar o efeito das instruções sobre o flag de CARRY, vamos supor que queremos subtrair o conteúdo do par DE do par HL, sem alterar o acumulador (ou registros B e C). Compare os seguintes programas:

FIGURA 3.15 - Simulação errada de SUB.

```
INC     A           3CH
DEC     A           3DH
SBC     HL,DE       ED52H
```

FIGURA 3.16 - Simulação correta de SUB.

```
ADD     A,0         C600H
SBC     HL,DE       ED52H
```

Eles têm significado? De fato, não, pois ninguém pode garantir qual o valor do CARRY no primeiro programa; portanto, ele poderá dar um resultado errado (no segundo programa, o CARRY será garantidamente 0 (zero) devido à instrução ADD A,0). Lembre-se que as instruções INC e DEC não afetam o CARRY...

Iremos falar agora sobre uma variável de sistema chamada VARTAB (F6C2H). Ela sempre aponta para o início da área de variáveis do programa BASIC, que é justamente o final do programa BASIC.

Assim fazendo:

```
PRINT PEEK (&HF6C2) + 256 * PEEK (&HF6C3)
```

teremos o endereço final do programa em BASIC e se subtrairmos 32767 (pois a área para o programa BASIC começa em 8000H (32768). Teremos o tamanho ocupado pelo programa BASIC.

Podemos fazer, então, a seguinte sub-rotina:

```
C000 11FF7F   LD    DE,7FFFH     ;aponta DE para
                                   área do BASIC
C003 2AC2F6   LD    HL,(0F6C2H)  ;aponta HL para o
                                   fim do programa
C006 C600     ADD   A,0          ;zera o CARRY
C008 ED52     SBC   HL,DE        ;subtrai DE de HL
C00A 7D       LD    A,L          ;carrega L em A
C00B 32F6F7   LD    (0F7F6H),A   ;carrega L em F7F6H
C00E 7C       LD    A,H          ;carrega H em A
C00F 32F7F7   LD    (0F7F7H),A   ;carrega H em F7F7H
C012 3E02     LD    A,2          ;carrega 2 em A
C014 3263F6   LD    (0F663H),A   ;carrega 2 em F663H
C017 C9       RET                ;retorna ao BASIC
```

Lembre-se que a instrução LD HL,(XXXXH) funciona da seguinte forma: coloca o conteúdo do endereço no registro L e o conteúdo do endereço (XXXXH+1) no registro H. Você deve estar, neste momento, com o programa MONITOR no micro. Utilize essa sub-rotina para saber quantos bytes ele ocupa!

RESUMO

Aprendemos a fazer adições e subtrações em L. M. e que o bit de CARRY do registro F serve para indicar "vai um" ou "menor que 0" p/ subtrações.

Abordamos também as instrucões INC e DEC que incrementam ou decrementam um registro ou um par de registros.

EXERCíCIOS

1- Escreva um programa em L. M. que retorne como valor inteiro 1 caso os endereços C200H e C201H forem iguais aos endereços C203H e C204H.

2- Faça um programa que deixe a flag de CARRY igual a 1 sem afetar nenhum registro ! (ou seja, deixando seus conteúdos iniciais iguais aos finais.). Obs- Faça isso sem utilizar a instrução ADD A,0.

3- Elabore um programa que some dois números de 4 bytes cada colocados nos bytes seguintes ao programa (sempre o menos significativo antes !). Para acessar a memória, use os pares DE e BC, apontando para o começo de cada número na memória, e utilize INC DE e INC BC para obter os bytes sucessivos! Utilize então o par HL como apontador do endereço no qual serão inseridos os resultados. Para incrementá-lo utilize a instrução INC HL.

INSTRUÇÕES VISTAS NESSE CAPíTULO

ADD A, registro
ADC A, registro
INC par de registro
ADC A,(HL)
SUB A,registro
SBC HL,par de registro
DEC par de registro
SBC A,(HL)
ADD HL, par de registro
INC registro
ADD A,(HL)
INC (HL)
SBC A,registro
DEC registro
SUB A,(HL)
DEC (HL)

4

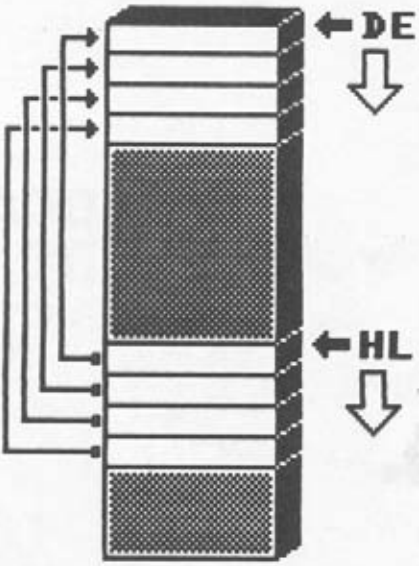
DESLOCAMENTO DE BLOCOS

Deslocar blocos significa deslocar de uma só vez os conteúdos de uma grande quantidade de bytes. Suponha que você tivesse uma sub-rotina razoavelmente grande (por exemplo, de 200 bytes), colocada a partir do endereço C000H e você quisesse transferí-la para começar no endereço B000H. Se você tivesse bastante paciência, poderia fazer como mostra a figura 4.1.

FIGURA 4.1 - Programa para a transferência de blocos.

```
C000 1100B0    LD    DE,0B000H    ;carrega DE com
                                    B000H (destino)
C003 2100C0    LD    HL,0C000H    ;carrega HL com
                                    C000H (fonte)
C006 7E        LD    A,(HL)       ;transfere fonte
C007 12        LD    (DE),A        para destino
C008 23        INC   HL           ;incrementa ponteiros
C009 13        INC   DE           ;
C00A 7E        LD    A,(HL)       ;transfere fonte
                                    para destino
C00B 12        LD    (DE),A
C00C 23        INC   HL           ;incrementa ponteiros
C00D 13        INC   DE
C00E 000000              .
C011 000000              .
C014 000000              .
```

... e assim por diante, 200 vezes.



Isto pode ser amenizado parcialmente com a instrução LDI (LoaD with Increment), que faz exatamente o que as quatro instruções que teríamos de repetir indefinidamente fazem, sem, no entanto, alterar o acumulador; ou seja, ela transfere o conteúdo da memória endereçada por HL ao conteúdo da memória endereçada por DE e, a seguir, incrementa os pares DE e HL e decrementa o par BC (mais tarde você perceberá a utilidade disso). Note que essa instrução só é válida para transferir dados de bytes endereçados por HL e DE. Por exemplo, a transferência dos bytes endereçados por HL para os endereçados por BC não pode ser feita numa única instrução.

O código de LDI é EDA0H; assim, o programa anterior poderia ser reescrito da seguinte forma:

```
1100B0  LD    DE,0B000H ;carrega ponteiros
2100C0  LD    HL,0C000H ;na memória
EDA0    LDI             ;transfere e
EDA0    LDI             ;incrementa ponteiros
```

... e assim por diante 200 vezes.

Isto reduz bastante o programa, mas há ainda uma maneira melhor de fazê-lo, usando a instrução LDIR (onde R significa Repeat). Essa instrução executa repetidamente o que a LDI faz até que o valor do par BC seja zero (aqui podemos ver a utilidade do decremento de BC pela instrução LDI). Assim se quisermos mover 200 bytes, deveríamos fazer:

```
C000 01C800  LD    BC,200      ;carrega o contador
C003 1100B0  LD    DE,0B000H ;carrega ponteiros
C006 2100C0  LD    HL,0C000H
C009 EDB0    LDIR              ;transfere,
C00B 00      NOP
```

Existem também as instruções LDD (código EDA8H) e LDDR (código EDB8H) que, em vez de incrementarem os pares de registros DE e HL, os decrementam (LDD = LoaD with Decrement). Nenhuma destas instruções afeta o valor da flag de CARRY.

TRANSFERÊNCIA ENTRE A RAM E A VRAM

Da mesma forma que existem as rotinas WRTVRM e RDVRM para gravar e ler um byte na VRAM, simulando uma instrução em Linguagem de Máquina. Existem mais duas rotinas LDIRVM (005CH) e LDIRMV (0059H) que se encarregam de transferir blocos entre a RAM e a VRAM, visto que o Z-80 não tem acesso direto aos bytes da VRAM.

A rotina LDIRVM se encarrega de copiar uma área da RAM na VRAM e a rotina LDIRMV copia uma área da VRAM na RAM.

Os pares de registradores deverão conter:

HL ——— endereço inicial do bloco a ser transferido.
DE ——— endereço de DEstino do bloco a ser transferido.
BC ——— número de Bytes a serem Copiados.

Tanto para uma como para outra rotina observe que a função de cada par de registros é a mesma que para a instrução LDIR.

As rotinas devem ser chamadas com o comando:

CALL (código CDH)

seguido do endereço da rotina (não se esqueça de inverter os 2 bytes do endereço).

Vejamos, então, um exemplo prático: quando trabalhamos apenas com o gravador cassete não é possível gravarmos os bytes da VRAM com um simples comando.

Se formos usar a função VPOKE do BASIC, o processo ficaria muito lento.

Então, porque não fazermos uma rotina em Linguagem de Máquina que receba como parâmetro um en-

dereço (na forma inteira) e copie da VRAM para a RAM 255 bytes a partir daquele endereço. A rotina retornaria uma STRING de 255 bytes.

Efetuando esta rotina um certo número de vezes poderemos gravar qualquer parte da VRAM em um arquivo com maior velocidade.

Veja na figura 4.2 como ficaria o programa.

Note que a sub-rotina LDIRMV do BIOS funciona de forma muito semelhante à instrução LDIR do Assembly Z-80. Como a transferência é da RAM para a VRAM, HL (origem) pode apontar qualquer endereço entre 0 e FFFFH, e DE entre 0 e 3FFFH.

FIGURA 4.2 - Transpondo blocos de memória

```
C000 2AF8F7    LD    HL,(0F7F8H) ;carrega HL com en-
                                   dereço do inicio
                                   da VRAM
C003 01FF00    LD    BC,0FFH     ;carrega BC com o
                                   nº de bytes
C006 110080    LD    DE,8000H    ;carrega DE com en-
                                   dereço de DEstino
                                   na RAM
C009 CD5900    CALL  LDIRMV      ;efetua a transfe-
                                   rência
C00C 3E03      LD    A,3         ;informa variável
                                   string ao BASIC
C00E 32F8F7    LD    (0F7F8H),A  ;
C011 21FF00    LD    HL,0FFH     ;aponta o endereço
C014 22F8F7    LD    (0F7F8H),HL ;F7F8H para o des-
                                   critor da string
C017 C9        RET               ;retorna ao BASIC
C018 FF        DEFB  0FFH        ;descritor da string
C019 21        DEFB  021H        ;endereço do 1º ca-
                                   ractere da string
C01A C0        DEFB  0C0H
C01B 000000          -
C01E 000000          -
```

O programa em BASIC teria o seguinte formato:

```
10 DEFUSR=&HC000
20 OPEN "CAS:TELA" FOR OUTPUT AS #1
30 FOR L%=0 TO 16389 - 255 STEP 255
40   A$ = USR(L%)
50   LINE PRINT #1,A$
60 NEXT
70 END
```

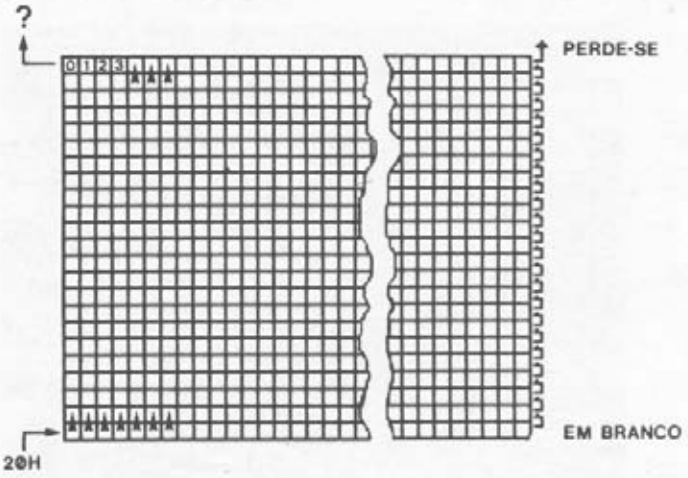
ROLANDO A TELA

Quando estamos trabalhando com as telas de texto (SCREEN 0 ou 1) existe o efeito de SCROLL que é executado automaticamente pelo micro quando a tela es-

tá cheia e é ordenada a impressão de uma linha. O SCROLL é executado antes da impressão, fazendo sumir a 1ª linha e copiando a 2ª na 1ª, a 3ª na 2ª e assim sucessivamente até o fim da tela e deixando a última linha em branco. Observe a figura 4.3 .

Uma possível maneira de se executar o SCROLL seria copiarmos os bytes da 2ª até a 24ª linha na RAM e logo após copiarmos esse bytes de volta à VRAM. Porém eles seriam jogados a partir do 1º byte da 1ª linha.

FIGURA 4.3 - Diagrama do SCROLL



Vejamos na figura 4.4 como ficaria o programa de SCROLL.

FIGURA 4.4 - Programa SCROLL.

```
C000 212800    LD    HL,0028H    ;transfere para C100H
                                  os 920 bytes
C003 110000    LD    DE,C100H    ;a partir do endereço
                                  28H(2ª linha)da VRAM
C006 019803    LD    BC,0398H    ;
C009 CD0000    CALL  LDIRMV      ;
C00C 2100C1    LD    HL,0C100H   ;transfere para a 1ª
                                  linha da VRAM
C00F 110000    LD    DE,0        ;920 bytes a partir
                                  de C100H da RAM
C012 019803    LD    BC,0398H
C015 CD0000    CALL  LDIRVM
C018 C9        RET
```

Execute essa rotina com o programa em BASIC da figura 4.5 .

FIGURA 4.5 - Programa BASIC.

```
10 DEFUSR=&HC000
20 FOR L=0 TO 23
30   A = USR(0)
40 NEXT L
50 END
```

Se tudo foi feito corretamente você terá o resultado da figura 4.6 .

FIGURA 4.6 - Tela após execução do programa SCROLL.

```
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
color   auto    goto    list    run
```

O que falta no programa SCROLL (figura 4.4) é a limpeza da última linha do vídeo.

Isso pode ser feito com o uso da rotina FILVRM, que preenche uma área da VRAM com um determinado valor. Essa rotina pede que o par HL contenha o endereço de início; o par BC o número de bytes; e que o registro A contenha o valor a ser preenchido. É importante ressaltar que essa rotina altera o valor de todos os registros.

Para que o programa de SCROLL fique correto devemos, então, inserir os comandos de modo que ele fique como mostra a figura 4.7 .

Execute o programa e veja a diferença. Veremos agora como fazer o efeito contrário, o chamado anti-SCROLL, que rola a tela no sentido inverso.

A idéia básica do programa é a mesma, só que ao invés de transferirmos o bloco da 2ª linha até a

última para a primeira linha, iremos transferir o bloco da 1ª à 23ª linha a partir da 2ª linha.

Observe o esquema da figura 4.8.

Note que usamos a RAM como uma espécie de buffer, pois não existe uma rotina no BIOS que faça a transferência entre blocos de memória dentro da própria VRAM.

O programa de anti-SCROLL ficaria como mostra a figura 4.9 .

FIGURA 4.7 - Programa SCROLL aperfeiçoado

```
C000 212800     LD    HL,28H
C003 1100C1     LD    DE,0C100H
C006 019803     LD    BC,398H
C009 CD5900     CALL  LDIRMV
C00C 2100C1     LD    HL,0C100H
C00F 110000     LD    DE,0
C012 019803     LD    BC,398H
C015 CD5C00     CALL  LDIRVM
C018 212800     LD    HL,40          ;preenche com
                                      40 espaços
C01B 012800     LD    BC,40          ;a última linha
                                      da tela
C01E 3E20       LD    A,20H          ;(a partir de 370H
                                      ;da VRAM)
C020 CD5600     CALL  FILVRM
C023 C9         RET
```

FIGURA 4.8 - Esquema para o anti-SCROLL.

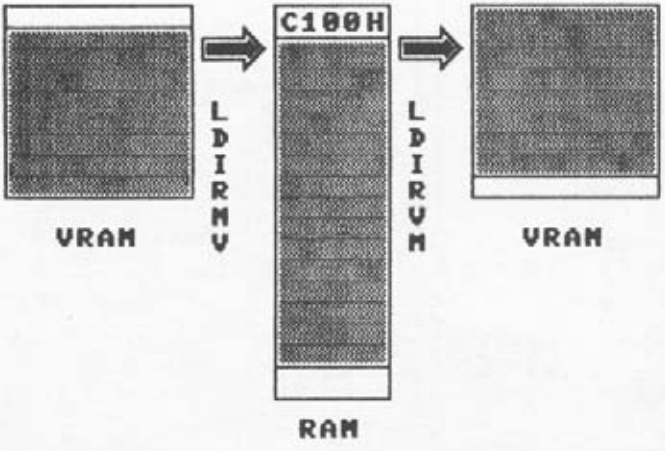


FIGURA 4.9 - Programa anti-SCROLL.

```
C000 210000    LD    HL,0
C003 1100C1    LD    DE,0C100H
C006 019803    LD    BC,398H
C009 CD0000    CALL  LDIRMV          ;transfere
C00C 2100C1    LD    HL,0C100H       ;para a RAM
C00F 111C00    LD    DE,28
C012 019803    LD    BC,398H
C015 CD0000    CALL  LDIRVM          ;transfere
C018 210000    LD    HL,0            ;para a VRAM
C01B 012800    LD    BC,28H
C01E 3E20      LD    A,20H
C020 CD0000    CALL  FILVRM          ;preenhe a 1ª
C023 C9        RET                   ;linha com
                                     ;espaços
```

Já que o anti-SCROLL libera a 1ª linha que tal incrementarmos o programa de modo que o próximo PRINT seja feito nela?

Isso é muito fácil. Basta chamar a rotina POSIT (00C6H) que posiciona o cursor em uma determinada linha ou coluna.

O registro H deve conter a coluna e o registro L a linha desejada.

A rotina POSIT modifica o valor do par AF. Com essa modificação o programa ficaria como exemplificado na figura 4.10 .

FIGURA 4.10 - Anti-SCROLL com LOCATE.

```
C000 210000          LD    HL,0
C003 1100C1          LD    DE,0C100H
C006 019803          LD    BC,398H
C009 CD5900          CALL  LDIRMV
C00C 2100C1          LD    HL,0C100H
C00F 112800          LD    DE,28H
C012 012800          LD    BC,28H
C015 CD5C00          CALL  LDIRVM
C018 210000          LD    HL,0
C01B 012800          LD    BC,28H
C01E 3E20            LD    A,20H
C020 CD5600          CALL  FILVRM
C023 2601            LD    H,1       ;Posiciona o cursor
C025 2E01            LD    L,1       ;na coluna 1 e
                                     ;na linha 1
C027 CDC600          CALL  POSIT
C02A C9              RET
005C          LDIRVM:EQU   005CH
0059          LDIRMV:EQU   0059H
0056          FILVRM:EQU   0056H
00C6          POSIT: EQU   00C6H
```

RESUMO

Neste capítulo aprendemos a deslocar blocos de memória usando as instruções LDI, LDIR, LDD e LDDR que utilizam como contador o par BC e como ponteiros para a memória os pares HL e DE. Essas instruções não afetam nem o acumulador nem a flag de CARRY.

Vimos as rotinas LDIRMV e LDIRVM do BIOS que transferem blocos da RAM para a VRAM e vice-versa; e a rotina FILVRM que preenche uma região da VRAM com um determinado valor.

Apresentamos um programa que permite gravarmos trechos da VRAM em cassete com maior velocidade.

EXERCíCIOS

1- Faça um programa para fazer um SCROLL apenas da metade superior da tela, deixando o resto inalterado. Coloque-o a partir do endereço C000H.

2- Faça um programa para fazer um anti-SCROLL apenas da metade inferior da tela, deixando o resto inalterado. Coloque-o a partir do endereço C0H0H.

3- Faça um programa que transfira 255 bytes de uma string (na RAM) para a VRAM e que retorne para uma variável inteira o último endereço na VRAM no qual foi inserido um dado.

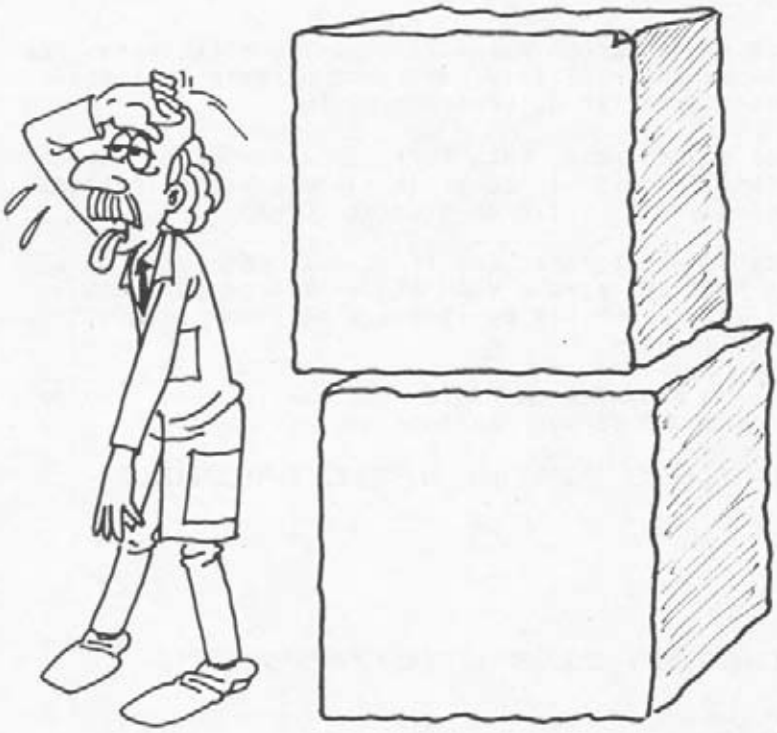
4- Faça um programa em BASIC para controlar o programa do exercício 3 de modo análogo ao da figura 4.5 .

INSTRUÇõES VISTAS NESTE CAPíTULO

LDI
LDIR
LDD
LDDR

ROTINAS DO BIOS UTILIZADAS

LDIRVM
LDIRMV
FILVRM
POSIT



5

SALTOS E SUB-ROTINAS

O STACK POINTER E O PROGRAM COUNTER

No capítulo 3 falamos sobre a flag de CARRY que fazia parte de um registro interno do microprocessador chamado F. Vamos agora estudar mais detalhadamente esse registro e dois outros muito importantes, chamados:

SP (Stack Pointer)
PC (Program Counter)

Estes dois registros têm 16 bits e, ao contrário dos já conhecidos pares BC, DE e HL, não podem ser divididos em dois registros de 8 bits.

FIGURA 5.1 - Registros internos do Z-80 .

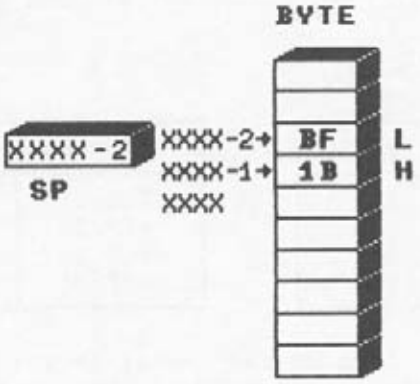
COMO "SALVAR" REGISTROS

A palavra stack significa pilha. Imagine uma caixa onde você coloca livros empilhados; o único livro que você poderá retirar facilmente é o que está no topo da pilha. Assim, se você colocar três livros A, B e C (nessa ordem), você só poderá retirá-los na ordem inversa, ou seja, C, B e A. Em outras palavras, o primeiro livro que você pode retirar da pilha foi o último a ser colocado. Desta mesma forma, o microprocessador usa uma área da memória RAM para empilhar dados, utilizando o registro SP (Stack Pointer = ponteiro da pilha) para indicar o endereço do primeiro byte da pilha, ou seja, o topo da pilha, onde está o último byte que foi colocado.

Uma das utilidades da "pilha de dados" na memória RAM é "salvar" momentaneamente o conteúdo dos registros quando necessitamos utilizá-los para alguma outra função. Por exemplo, suponha que num dado ponto do programa você precise utilizar os registros mas que eles estejam todos ocupados com dados que você não deseja perder. Uma alternativa é usar uma região da RAM para guardar momentaneamente o conteúdo dos registros necessários e, após eles terem sido usados, buscar de volta na RAM o seu antigo valor. Para tanto, existe uma, instrução para colocar pares de registros no topo da pilha, chamada PUSH, e outra para retirá-los, chamada POP. Por exemplo, imagine que o par HL contenha o número 1B8FH e que o registro SP contenha o número XXXXH.

A instrução PUSH HL coloca o conteúdo do registro H (1BH) no endereço XXXXH-1 (SP-1) e o registro L no endereço XXXXH-2 (SP-2). Com isso o registro SP passa a indicar XXXXH-2. Observe que essa instrução, assim como a LD, não altera o par HL, apenas copia seus valores na memória. Uma vez copiados, podemos utilizar H e L para outras finalidades e, feito isso, obter de volta seus antigos valores fazendo POP HL, e SP voltará a indicar XXXXH. Essas instruções só funcionam para pares de registros; você não pode fazer PUSH B ou POP E, por exemplo! Note que, dada a situação inicial, ou seja, antes de fazer o PUSH (SP = XXXXH), se em vez de PUSH HL tivéssemos feito POP HL, o conteúdo do par HL seria perdido e substituído pelo que estivesse nos bytes XXXXH (registro L) XXXXH+1 (registro H), passando SP a indicar XXXXH+2.

FIGURA 5.2 - Resultado de um PUSH HL.



Assim como nos loops FOR/NEXT, quando colocamos um loop dentro do outro, cuidados semelhantes devem ser tomados quando fizermos vários PUSHs. Portanto, se você fizer PUSH HL e, a seguir, PUSH BC, quem ficará no topo da pilha será o par BC; logo, para ter os valores originais de volta, você deve fazer primeiro POP BC e, depois, POP HL. Se você esquecer essa inversão, ocorrerá uma troca, ou seja, o que estava no par HL passa para o par BC e vice-versa. Da mesma forma, se com um único PUSH e um único POP você errar os pares, por exemplo, PUSH HL e, a seguir, POP BC, o que estava no par HL será copiado no par BC (o que equivale a fazer LD B,H e LD C,L).

É muito importante que, em um programa em Linguagem de Máquina, o número de POPs seja igual ao número de PUSHs, pois o Interpretador guarda alguns valores importantes no STACK e se ao devolver o comando do micro, houver alguma diferença o Interpretador pode perder o controle obrigando-o a dar um RESET ou a desligar e ligar a máquina.

O STACK POINTER deve ter como seu valor inicial um endereço bem no fim da memória para evitar que os daods empilhados atinjam os programas. Assim, por exemplo, se SP = C100H e tivermos um programa que termina em C008H com mais de 20 PUSHs antes que haja POP, o SP passa a indicar regiões de memória onde está o fim do programa. Portanto, você estará alterando o seu próprio programa, o que poderá ocasionar consequências não desejáveis... (um belo quadro de arte moderna, por

exemplo). O valor inicial do SP fica a cargo do programa interpretador.

Estão apresentados na figura 5.3 os códigos das instruções PUSH e POP.

FIGURA 5.3 - Instruções PUSH e POP.

INSTRUÇÃO	CÓDIGO
PUSH AF	F5H
PUSH BC	C5H
PUSH DE	D5H
PUSH HL	E5H

INSTRUÇÃO	CÓDIGO
POP AF	F1H
POP BC	C1H
POP DE	D1H
POP HL	E1H

O par AF é um par peculiar que só pode ser utilizado por ocasião de PUSH e POP; ele consta do acumulador e do registro F, que contém as flags (entre elas a já conhecida flag de CARRY). As instruções PUSH e POP não alteram a flag de CARRY.

Para exemplificar, suponha que você queira somar 25 número que foi passado como parâmetro sem alterar nenhum outro registro, nem mesmo o registro A; assim, você deve "salvar" A, o que pode ser feito usando PUSH AF. Portanto, basta fazer como mostra a figura 5.4.

FIGURA 5.4 - Soma preservando o acumulador e a flag de CARRY.

```
C000 E5        PUSH HL            ;salva HL no
                                    topo da pilha
C001 F5        PUSH AF            ;salva AF no
                                    topo da pilha
C002 21F8F7    LD   HL,0F7F8H     ;aponta HL para
                                    o byte LSB
C005 7E        LD   A,(HL)        ;carrega o valor
                                    em A
C006 C619      ADD  A,19H         ;adiciona 25 em A
C008 77        LD   (HL),A        ;coloca o resul-
                                    tado em F7F8H
C009 23        INC  HL            ;aponta HL para
                                    o byte MSB
C00A 7E        LD   A,(HL)        ;carrega o valor
                                    em A
C00B CE00      ADC  A,0           ;soma o CARRY
C00D 77        LD   (HL),A        ;coloca o resul-
                                    tado em F7F8H
C00E F1        POP  AF            ;recupera AF
C00F E1        POP  HL            ;recupera HL
C010 C9        RET
```

Perceba que nem a flag de CARRY será alterada apesar da instrução ADD, pois fazendo PUSH AF você "salvou" o valor do acumulador e de todas as flags.

Da mesma maneira que nós utilizamos os outros pares de registros, podemos fazer "contas" com o conteúdo de SP ou carregar valores no mesmo. Assim, temos as instruções:

FIGURA 5.5 - Instruções que utilizam diretamente o SP.

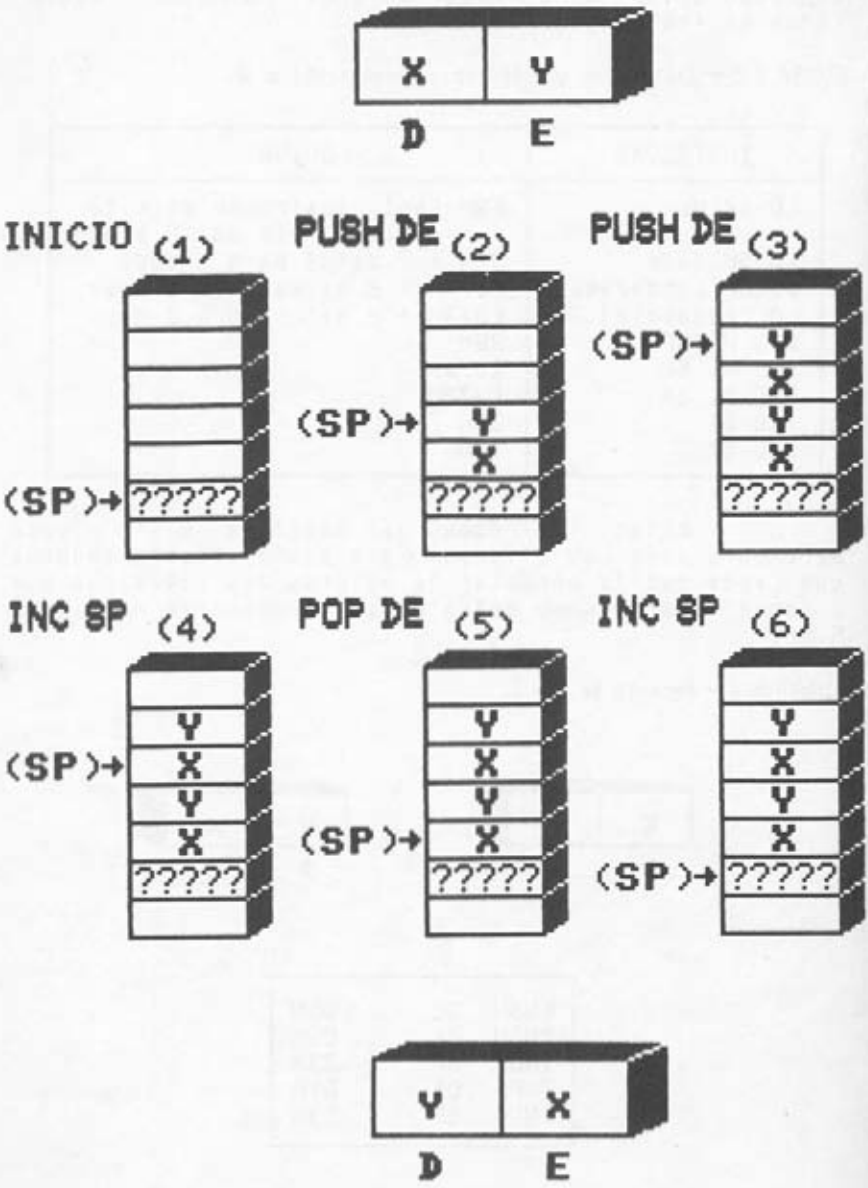
INSTRUÇÃO	CÓDIGO
LD SP,HL	F9H (esta instrução permite cópia direta de 16 bits)
LD SP,dado	31H + 2 bytes para o dado
LD SP,(endereço)	ED7BH + 2 bytes para o dado
LD (endereço),SP	ED73H + 2 bytes para o dado
ADD HL,SP	39H
ADC HL,SP	ED7AH
SBC HL,SP	ED72H
INC SP	33H
DEC SP	3BH

Estas instruções são bastante úteis e você perceberá isso com o tempo. Para exemplificar, suponha que você queira permutar os valores dos registros D e E sem alterar nenhum outro registro; observe na figura 5.6 .

FIGURA 5.6 - Permuta de D e E.


```
PUSH  DE      D5H
PUSH  DE      D5H
INC   SP      33H
POP   DE      D1H
INC   SP      33H
```

FIGURA 5.7 - Visualização de DE e do "stack".



O último INC SP é necessário para fazer com que o SP volte a seu valor original. Vamos testar esse procedimento. Execute o programa da figura 5.8 que troca o valor do byte LSB com o MSB.

FIGURA 5.8 - Permuta de 2 bytes.

```
C000 2AF8F7   LD    HL,(0F7F8H) ;carrega HL com
                                  o valor
C003 E5       PUSH  HL          ;
C004 E5       PUSH  HL          ;
C005 33       INC   SP          ;troca H com L
C006 E1       POP   HL          ;
C007 33       INC   SP          ;
C008 22F9F7   LD    (0F7F9H),HL ;recupera o valor
                                  na memória
C00B C9       RET
```

Teste o programa passando, como parâmetros, números em hexadecimal pois eles também são armazenados na forma inteira. Chame a rotina da seguinte maneira.

```
PRINT HEX$(USR(número))
```

Outra utilidade do SP é permitir a utilização de sub-rotinas em Linguagem de Máquina; veremos isso mais adiante neste mesmo capítulo.

DESVIOS ABSOLUTOS E RELATIVOS

O PC (Program Counter = Contador de programa) é outro registro interno de 16 bits que não pode ser divido e cuja função é indicar ao microprocessador em que endereço da memória ele deve "buscar" a instrução em Linguagem de Máquina. Assim, ao fazer:

```
DEF USR = &HC000 : PRINT USR (0)
```

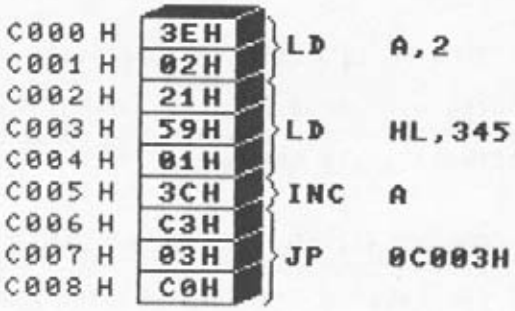
O número ou variável e seu tipo são armazenados nos endereços padrão e em seguida o PC é carregado com C000H e o microprocessador executa a instrução no endereço C000H e incrementa o valor do PC um número de vezes que corresponde ao número de bytes da instrução para saber o endereço da próxima instrução a ser processada. Por exemplo, se a primeira instrução for LD B,A que tem um byte, ela será executada e PC passará a indicar C001H; se a segunda instrução for

LD A,2 (2 bytes), ela será executada e PC indicará C003H, e assim por diante.

Você deve ter notado então que se pudermos alterar o valor do PC teremos o mesmo efeito que a instrução GOTO em BASIC! Mas lembre-se que em Linguagem de Máquina não temos etiquetas numeradas para as linhas de programa e, portanto, devemos ir para endereços da memória (muito cuidado deve ser tomado para não irmos para o meio de uma instrução). Assim, temos a instrução JP (Jump = salte). Por exemplo, JP 0C000H carregará PC com o endereço C000H (sem, no entanto, incrementá-lo) e o microprocessador "pensará" que C000H é o endereço da próxima instrução do programa.

Note que se tivermos um programa como mostra a figura 5.9 .

FIGURA 5.9 - Visualização de memória.



é provável que "coisas estranhas" aconteçam, pois estamos indo com JP 0C003H para um endereço que não é o começo de nenhuma instrução, e o microprocessador, não sabendo disso, executará fielmente a ordem e tentará interpretar a memória C003H (cujo conteúdo é 59H) como o começo de uma instrução, o que certamente causará "coisas malucas"!

Ainda usando o exemplo anterior, vamos substituir o nosso JUMP por JP 0C000H. Veja que, apesar de agora esse endereço fazer sentido, acabamos de criar um loop infinito, de onde o MSX nunca sairá nem mesmo usando CTRL+STOP. O único remédio é desligar o computador, ou usar o WARM START.

O código da instrução JUMP é:

JP endereço C3H + 2 bytes para endereço

ou, usando o endereço dado pelo par HL:

JP (HL) código E9H

Note que agora fica claro, porque, além de colocarmos os mnemônicos e seus códigos, devemos colocar também os endereços do começo de cada instrução, pois eles, além de mostrarem onde colocar variáveis, facilitam o uso das instruções de JUMP! Portanto, veja como é importante saber quantos bytes tem cada instrução para podermos "contar" certo os endereços...

Uma outra instrução de "salto" similar à JP, é a JR (Jump Relative = salto relativo). Essa instrução significa "salte para frente ou para trás um determinado número de bytes" (127 para frente ou 128 para trás no máximo!), somando o número dado ao endereço imediatamente após o final da instrução. Assim, JR 0 não tem efeito nenhum, JR 1 "pulará" o próximo byte, JR 2 "pulará" os dois próximos bytes e assim por diante! Novamente, cuidado para não cair no meio de instruções:

FIGURA 5.10 - Exemplo de utilização errada de JR.

```
C000 3E02          LD   A,2
C002 215901        LD   HL,345
C005 3C            INC  A
C006 C303C0        JP   0C003H
```

O código de JR é: 18H + 1 byte para o dado.

Surge agora uma pergunta: e para "pular" para trás? Devemos usar números negativos? Como representá-los só com 0's e 1's? Já havíamos dito no capítulo 3, quando falamos na subtração, que iríamos deixar os números negativos para mais tarde: chegou a hora! Na verdade é tudo uma questão de convenção: você interpreta os 8 bits de um registro como lhe convier: ou você só pensa em números positivos, onde você poderá representar desde 0 até 255, ou você pensa em números positivos e negativos, podendo então representar desde -128 até 127 (note que continuam sendo 256 números!).

A convenção usada em Linguagem de Máquina é a seguinte:

- número zero: 0000 0000B

- números positivos: de 0000 0001B (número + 1) 01H
  até 0111 1111B (número + 127) F0H

- números negativos: pegue o número positivo correspondente em binário, troque os 0's pelos 1's e some um ao resultado! (Pode parecer, à primeira vista, uma convenção meio sem sentido, mas se você algum dia estudar profundamente a Linguagem de Máquina, entenderá a vantagem disso!) Por exemplo, para obter o número -2 :

```
                +2 =  0000 0010B
bits invertidos ---->  1111 1101B
                    +         1B
                    ------------
                       1111 1110B
                        -2 =  FEH
```

Você pode obter a representação dos números negativos com o programa da figura 5.11.

FIGURA 5.11 - Representação hexadecimal de números negativos.

```
10 FOR L = -1 TO -128 STEP -1
20   POKE &HFFCAH,L
30   PRINT L;"...";HEX$(PEEK(&HFFCA))
40 NEXT L
50 END
```

De qualquer forma, para facilitar vamos apresentar uma tabela (figura 5.12) com os números negativos de -1 a -128.

Para utilizá-la, junte o dígito da linha horizontal com o dígito da coluna vertical. Por exemplo:

-55 = C9H = +201
-78 = B2H = +178

Preste bem atenção agora: a instrução JR sempre soma o seu dado ao endereço do primeiro byte que fica após o término da instrução (com números positivos o raciocínio é fácil e já foi mencionado; com

números negativos você sempre deve subtrair dois a mais do que parece à primeira vista, pois JR é uma instrução de 2 bytes). Assim, fazer a instrução JR -2 (JR 0FEH) significa fazer um loop infinitivo, pois você estará sempre voltando ao começo dela mesma. Veja um exemplo de como usar corretamente JR com números negativos: suponha que você tenha a instrução no endereço C002H e deseja voltar para o endereço C000H; então, C002 JR -4 18FCH

FIGURA 5.12 - Números negativos de 1 byte.

	F	E	D	C	B	A	9	8
F	-001	-017	-033	-049	-065	-081	-097	-113
E	-002	-018	-034	-050	-066	-082	-098	-114
D	-003	-019	-035	-051	-067	-083	-099	-115
C	-004	-020	-036	-052	-068	-084	-100	-116
B	-005	-021	-037	-053	-069	-085	-101	-117
A	-006	-022	-038	-054	-070	-086	-102	-118
9	-007	-023	-039	-055	-071	-087	-103	-119
8	-008	-024	-040	-056	-072	-088	-104	-120
7	-009	-025	-041	-057	-073	-089	-105	-121
6	-010	-026	-042	-058	-074	-090	-106	-122
5	-011	-027	-043	-059	-075	-091	-107	-123
4	-012	-028	-044	-060	-076	-092	-108	-124
3	-013	-029	-045	-061	-077	-093	-109	-125
2	-014	-030	-046	-062	-078	-094	-110	-126
1	-015	-031	-047	-063	-079	-095	-111	-127
0	-016	-032	-048	-064	-080	-096	-112	-128

Junte a coluna com a linha. Ex: -55 = C9

DESVIOS CONDICIONAIS

As instruções JP e JR, usadas desta maneira, são chamadas saltos incondicionais. Assim como no BASIC a instrução GOTO ganha bastante poder se usada juntamente com a instrução IF/THEN, temos em linguagem de máquina algo equivalente: saltos condicionais, ou seja, "salte se ocorrer uma dada condição como resultado de uma operação". Embora não tenhamos a mesma flexibilidade que o IF/THEN, há quatro condições que podem ser testadas usando JR e oito usando JP . Duas delas se baseiam no CARRY: "salte se o CARRY for 1" e "salte se o CARRY for zero".

```
JR   C,dado        (38H+1 byte para dado)
                   (Jump Relative if Carry = 1)
JR   NC,dado       (30H+1 byte para dado)
                   (Jump Relative if No Carry)
JP   C,endereço    (0DAH+2 bytes para endereço)
JP   NC,endereço   (0D2H+2 bytes para endereço)
```

Para poder entender as outras condições, vamos antes estudar um pouco mais o registro das flags (F) que, como sabemos, tem 8 bits, sendo um deles para indicar a flag de CARRY.Vamos agora ver mais algumas flags.

Temos uma flag de paridade (P) que assinala se o número de 1s ou 0s do acumulador é par (EVEN) ou ímpar (ODD) após efetuada uma operação lógica (ver apêndice 2).Assim temos:

```
JP   PE,endereço   (0EAH + 2 bytes para
                    endereço)
                   (JumP if Parity is Even)
JP   PO,endereço   (0E2H + 2 bytes para
                    endereço)
                   (JumP if Parity is Odd)
```

Além disto, essa mesma flag serve para testar se houve OVERFLOW (flag O).que seria uma espécie de CARRY para quando trabalhamos com números positivos e negativos. Entretanto ela é apenas uma flag de alarme, não permitindo identificar qual o resultado real (como era possível com o CARRY usando ADC ou SBC). De fato, quando calculamos usando a convenção de números negativos, podem haver trocas acidentais de sinais; por exemplo, se você somar 4AH, que é positivo, com

44H, que também é positivo, você obterá 8EH, que é negativo, pois agora o maior número positivo representável em um byte é 127 (7FH). Note que você terá OVERFLOW mas não CARRY = 1, pois usando a convenção de números apenas positivos, esta soma produz CARRY = 0. Desta forma, essa flag funciona de maneira que JP PE equivale também a JUMP, se houve OVERFLOW, e JP PO equivale a JUMP, se não houve OVERFLOW. Não existe salto relativo usando essa flag como condição. Note que por ser uma flag dupla, algumas instruções a afetam como sendo P (paridade) e outras como O (OVERFLOW).

Outra flag é a Z, que indica se o resultado de uma operação é zero. Assim, temos:

JR Z,dado	(28H + 1 byte para dado) (Jump Relative if Zero)
JR NZ,dado	(20H + 1 byte para dado) (Jump Relative Not Zero)
JP Z,endereço	(0CAH + 2 bytes para endereço)
JP NZ,endereço	(0C2H + 2 bytes para endereço)

A última flag que estudaremos neste capítulo é uma flag de sinal (S), que também só tem sentido em ser usada quando utilizamos a convenção de números negativos. Dessa forma, você pode "saltar" dependendo do resultado de uma operação (ver apêndice 2) ser negativo ou positivo:

JP M,endereço	(0FAH + 2 bytes para endereço) (JumP if Minus)
JP P,endereço	(0F2H + 2 bytes para endereço) (JumP Plus)

Também não existe salto relativo para esta condição.

As flags ensinadas até agora estão localizadas no registro F como mostra a figura 5.13 .

Obviamente as instruções de JUMP não afetam nenhuma flag; mas note que, agora, não podemos nos preocupar somente com a flag de CARRY! Para cada instrução, devemos saber precisamente quais flags são afetadas. Por exemplo, as instruções de LD não afetam

nenhuma flag e as instruções de ADD, SUB, ADC e SBC afetam todas as flags. Procuraremos, na medida do possível, explicar as flags afetadas sempre que falarmos em alguma instrução; no entanto, convém sempre utilizar o apêndice 2, que possui uma tabela com todas as instruções, seus códigos hexadecimais e as flags afetadas. Cabe aqui uma observação importante quanto às instruções JP e JR: as instruções JP, por usarem endereços absolutos, têm a desvantagem de o endereço dever sempre ser recalculado quando houver alguma modificação no lugar em que está o programa. Assim, se você tiver um programa a partir do endereço C000H e quiser colocá-lo a partir do endereço C100H todos os endereços JUMPs terão de ser acrescidos de 100. Isto não acontece com os JUMPs relativos (JR), fato este que os torna facilmente relocáveis.

FIGURA 5.13 - As flags no registro F.

S - SIGN (1 negativo / 0 positivo)
Z - ZERO (1 zero / 0 not zero)
P/O - PARITY / OVERFLOW (1 Even / 0 Odd)
C - CARRY

Apenas para completar as instruções com números negativos, apresentaremos a instrução NEG, que muda o sinal do número que está no acumulador (seu código é ED44H). Se você quiser, por analogia às demais instruções, chamá-la de NEG A, não há problema, no nosso caso. Entretanto, para uso com programas que traduzem automaticamente os mnemônicos para os códigos hexadecimais (programas ASSEMBLER), é conveniente usar o formato padronizado, ou seja, NEG.

Execute o programa da figura 5.14 .

FIGURA 5.14 - Exemplo de uso do NEG.

```
C000 3AF8F7    LD    A,(0F7F8H)   ;carrega A com LSB
C003 ED44      NEG                ;inverte o sinal
C005 32F8F7    LD    (0F7F8H),A   ;carrega MSB
C008 3E02      LD    A,2          ;informa variável
C00A 3263F6    LD    (0F663H),A             inteira
C00D C9        RET
```

Com o seguinte comando:

```
PRINT HEX$ (USR (&H37))
```

você deverá obter como retorno: C937

Note que C9H equivale a -37H usando a notação de números negativos.

Como você deve ter notado, nas instruções de salto condicional nós sublinhamos a frase: resultado de uma operação. De fato, os JUMP's condicionais funcionam se, antes, alguma operação afetou as flags adequadamente. Por exemplo, não adianta carregar o acumulador com o conteúdo de algum registro ou de memória e, em seguida, querer usar o JUMP condicional baseado no conteúdo do acumuldor. Isso simplesmente utilizará, para testes, as flags da última operação que foi efetuada antes desse carregamento. Logo, se você quer "saltar" para algum ponto (C020H), baseado, por exemplo, no fato de um determinado endereço conter o número zero não adianta fazer o mostrado na figura 5.15 .

FIGURA 5.15 - Exemplo de como não usar JP condicional após LD.

```
C000 3A10C0    LD    A,(0C010H)  ;não afeta nenhuma
                                 FLAG
C003 2818      JR    Z,0C020H    ;testa FLAG gerada
```

Isso é válido para todos os saltos condicionais. Para poder testar condições, nesses casos, é necessário uma nova instrução, chamada CP (ComPare), que compara o conteúdo do acumulador com qualquer outro registro, com um dado número (entre 0 e 255 ou entre -128 e 127) ou um byte endereçado por HL. A instrução CP funciona da seguinte maneira: sem alterar nenhum registro, ela verifica qual seria o resultado do acumulador menos o registro ou dado comparado, atualizando as flags convenientemente. Assim, se o número comparado for igual ao acumulador, a flag de zero será setada (colocada em 1); se for menor que o acumulador, a flag de sinal indicará positivo (0) e, se for maior, negativo (1). Isso, aliado às instruções JP ou JR, equivale, em BASIC, a fazer:

$$\text{IF} \begin{Bmatrix} A = B \\ A > B \\ A < B \end{Bmatrix} \text{THEN GOTO XXXX}$$

Repare que, para testar se o acumulador é maior ou igual ou menor ou igual, duas flags devem ser testadas.

Assim, para que o exemplo anterior funcione, devemos fazer:

FIGURA 5.16 - Exemplo de uso correto de JP condicional após LD.

```
C000 3A10C0   LD   A,(0C010H)  ;insere conteúdo
                                C010H EM A
C003 FE00     CP   0           ;compara A com 0 e,
                                se for igual, FLAG
                                Z=1, se for dife-
                                rente, FLAG Z=0
C005 CA07C0   JP   Z,0C007H    ;pula para C007H se
                                FLAG Z=1, caso con-
                                trário, prossegue
```

Estes cuidados não são necessários após se efetuar uma operação aritmética ou lógica pois as flags são calculadas automaticamente (ver apêndice 2).

Obviamente a instrução CP afeta todas as flags. Os códigos estão na figura 5.17 .

FIGURA 5.17 - A instrução CP.

INSTRUÇÃO	CÓDIGO
CP dado	FEH + 1 byte para o dado
CP (HL)	BEH
CP A	BFH
CP B	B8H
CP C	B9H
CP D	BAH
CP E	BBH
CP H	BCH
CP L	BDH

Para finalizar os saltos condicionais, vamos aprender um "truque" que nos permite usar JR em vez de JP quando queremos comparar um número com o acumulador e saber se o mesmo é maior ou menor: se o número a ser comparado for maior, a diferença do acumulador (supondo-o positivo) menos o número resulta negativa e, portanto, gera CARRY = 1. Logo, apesar de não existir a instrução JR M, você pode usar JR C. Obviamente, a instrução inexistente JR P pode ser simulada por JR

NC. Naturalmente, isso só funcionará se adotarmos a convenção de números positivos ou, se adotarmos a outra, garantirmos que no acumulador esteja um número positivo. Perceba que a instrução JR NC considera 0 como número positivo (o mesmo é válido para JP NC e JP P).

Alguns autores sugerem o uso de outros nomes para as instruções de salto condicional quando elas não são usadas especificamente para teste de flags mas sim para teste de números; assim, JP Z (JumP if Zero) seria melhor escrita por JP E (JumP if Equal), JP M (JumP if Minus), por JP L (JumP if Less) e JP P (JumP if Positive) por JP G (JumP if Greater). Outros chegam até a sofisticação de "fundir" duas instruções em uma só:

```
JP GE (JumP if Greater or Equal JP Z * JP P)
JP LE (JumP if Less or Equal JP Z * JP M)
```

Apesar dessas instruções facilitarem a escrita dos programas, elas requerem um trabalho adicional quando são tradução dos mnemônicos, pois não constam nas tabelas, além de não funcionarem com os programas tradutores ASSEMBLER.

"LOOPS" EM LINGUAGEM DE MÁQUINA

Existe, em Linguagem de Máquina, uma instrução especial que facilita executar loops:

```
DJNZ dado (10H + 1 byte para o dado)
(Decrement B and Jump if Not Zero)
```

ela usa o registro B como contador e executa um salto relativo enquanto B não atinge zero.

Com a instrução DJNZ nós podemos melhorar o programa de SCROLL e anti-SCROLL do capítulo 4. Aqueles programas, funcionam muito bem, mas possuem uma grave falha. Como não é possível um LDIR ou um LDDR ou um LDDR dentro da própria VRAM, um bloco de memória é jogado da VRAM para a RAM e novamente para a VRAM (só que com outro endereço inicial). Para um SCROLL ou anti-SCROLL na SCREEN 0, temos que reservar 920 bytes da RAM para esse deslocamento o que é,dependendo das circunstâncias, um desperdício de memória.

Vejamos como ficaria o anti-SCROLL usando DJNZ.

FIGURA 5.18 - Anti-SCROLL com DJNZ.

```
C000 219803    LD    HL,920
C003 112800    LD    DE,40
C006 0618      LD    B,24
C008 C5        PUSH  BC
C009 0628      LD    B,40
C00B ED52      SBC   HL,DE
C00D CD4A00    CALL  4AH
C010 19        ADD   HL,DE
C011 CD4D00    CALL  4DH
C014 2B        DEC   HL
C015 1000      DJNZ  -10
C017 C1        POP   BC
C018 1000      DJNZ  -16
C01A 210000    LD    HL,0
C01D 012800    LD    BC,40
C020 3E20      LD    A,20H
C022 CD5600    CALL  56H
C025 210101    LD    HL,101H
C028 CDC600    CALL  0C6H
C02B C9        RET
```

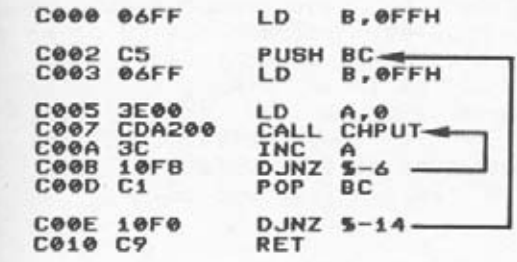
LOOPS DENTRO DE LOOPS

Quando estamos trabalhando em BASIC podemos facilmente elaborar um programa com loops dentro de loops. Veja a figura 5.19.

FIGURA 5.19 - Loops dentro de loops em BASIC.

```
10 FOR L = 0 TO 255
20   FOR B = 0 TO 255
30     PRINT L;"...";B
40   NEXT B
50 NEXT L
60 END
```

Em Linguagem de Máquina, quando fazemos loops com a instrução DNJZ o maior número de vezes que o loop é executado é 256. Se necessitamos de um número maior de vezes devemos proceder da seguinte maneira:



O segredo está em guardar o valor do par BC no STACK, preservando assim o valor do passo no loop externo e após executar o loop interno recuperar o valor armazenado no STACK com um PUSH BC (antes do segundo DJNZ).

SUB-ROTINAS

Como vimos nos capítulos anteriores, podemos chamar sub-rotinas que podem ser as sub-rotinas do BIOS ou as que inventarmos nos programas em Linguagem de Máquina.

A instrução equivalente ao GOSUB é a:

```
CALL endereço (CDH + 2 bytes para endereço)
```

e, para retornar ao programa principal, temos a já famosa RET (C9H). Assim como em BASIC seria possível uma estrutura do tipo

```
IF condição THEN GOSUB XXXX
```

também em linguagem de máquina existem instruções para chamar sub-rotinas condicionalmente. Veja a figura 5.20.

E, como era de se esperar, podemos também fazer RET condicionalmente. Veja a figura 5.21.

Como funcionam essas instruções? Ao fazer um CALL, o conteúdo do contador de programa que indica a próxima instrução (PC) é colocado no topo da pilha indicada pelo SP (seria como se existisse uma instrução PUSH PC) e ele é então carregado com o endereço que

está logo após o código de CALL, que é onde começa a sub-rotina. Assim o computador passa a executar as instruções de RET, o topo da pilha é recolocado no PC (como se fosse um POP PC); portanto, o programa volta para a próxima instrução após o CALL. Por exemplo, suponha que existisse, a partir do endereço C100H, uma sub-rotina para fazer uma soma de dois bytes da memória indicados por DE e HL. Veja a figura 5.22.

FIGURA 5.20 - Instrução CALL.

INSTRUÇÃO	CÓDIGO
CALL Z , endereço	CCH + 2 bytes para endereço
CALL NZ , endereço	C4H + 2 bytes para endereço
CALL C , endereço	DCH + 2 bytes para endereço
CALL NC , endereço	D4H + 2 bytes para endereço
CALL PE , endereço	ECH + 2 bytes para endereço
CALL PO , endereço	E4H + 2 bytes para endereço
CALL M , endereço	FCH + 2 bytes para endereço
CALL P , endereço	F4H + 2 bytes para endereço

FIGURA 5.21 - A instrução RET.

INSTRUÇÃO	CÓDIGO
RET Z	C8H
RET NZ	C0H
RET C	D8H
RET NC	D0H

INSTRUÇÃO	CÓDIGO
RET PE	E8H
RET PO	E0H
RET M	F8H
RET P	F0H

FIGURA 5.22 - Sub-rotina para somar dois conteúdos de memória.

```
C100 1A        LD    A,(DE)      ;carrega em A o
                                   conteúdo do en-
                                   dereço apontado
                                   por DE
C101 47        LD    B,A         ;transfere A para B
C102 7E        LD    A,(HL)      ;carrega em A o
                                   conteúdo do en-
                                   dereço apontado
                                   por HL
C103 80        ADD   A,B         ;adiciona B em A
C104 C9        RET               ;retorna ao BASIC
```

Assim, se porventura o valor de SP for alterado durante a sub-rotina, deveremos garantir que, antes da instrução RET, seu valor seja SP-2...Note que esta estrutura permite que haja sub-rotinas dentro de sub-rotinas. Muitas vezes, ao chamar uma sub-rotina, é necessário "salvar" os conteúdos de alguns registros, o que pode ser feito usando PUSHs. Lembre-se sempre de fazer os POPs correspondentes (na ordem certa) antes de fazer RET pois, caso contrário, provavelmente você obterá o nosso já famoso efeito "estranho".

Para exemplificar, vamos fazer uma sub-rotina que multiplica dois números positivos menores que 256, que estão no acumulador e no registro E , respectivamente. A sub-rotina deve colocar o resultado no par HL sem alterar nenhum outro registro. Naturalmente, como não temos uma instrução de multiplicação, faremos somas sucessivas. Usaremos um loop utilizando a instrução DJNZ que, como vimos anteriormente, funciona da seguinte maneira: decrementa o registro B e, se o resultado for diferente de zero, faz um salto relativo (+ 127 a -128) para o endereço indicado, caso contrário, prossegue o programa. Portanto, o registro B funciona como um contador e será alterado na sub-rotina devendo então ser "salvo" (PUSH BC). Coloque a sub-rotina a partir do endereço C100H e tente colocar os códigos em Linguagem de Máquina (utilize o apêndice 2 para ajudá-lo).

FIGURA 5.23 - Sub-rotina que multiplica números.

```
PUSH BC          ;salva BC
LD   B,E         ;coloca um dos operandos
                  em B
PUSH DE          ;salva DE
LD   D,0         ;coloca o outro operando
                  em DE
LD   E,A         ;
LD   HL,0        ;insere 0 em HL
ADD  HL,DE       ;soma o 2º operando
                  sucessivamente tantas
                  vezes quantas indicar o
                  1º operando (B)
DJNZ -3
POP  DE          ;restabelece os valores
                  iniciais dos registros
POP  BC
RET
```

Perceba que os PUSHs e POPs permitiram a condição de não alterar nenhum outro registro. Veja

que o número de PUSHs é igual ao número de POPs e, com isto, o valor de SP não é modificado, fazendo com que a sub-rotina volte para o endereço certo. Vamos tentar utilizá-la? Façamos um programa, a partir do endereço C000H, que multiplique dois números (por exemplo, 15 e 24), usando a sub-rotina da figura 5.24.

FIGURA 5.24 - Programa principal que chama a sub-rotina.

```
C000 3E0F      LD    A,0FH          ;carrega A com 15
C002 1E18      LD    E,18H          ;carrega E com 24
C004 CD00C1    CALL  0C100H         ;chama sub-rotina de
                                    multiplicação
C007 2AF8F7    LD    HL,(0F7F8H)    ;prepara saída
C00A 3E02      LD    A,2            ;carrega A com 2
C00C 3263F6    LD    (0F663H),A     ;indica valor
                                     inteiro ao BASIC

C00F C9        RET
```

Verifique se o resultado é 360, caso contrário disassemble o programa com o MONITOR e tente achar o erro.

RESUMO

Neste capítulo vimos dois novos registros internos do microprocesador os quais têm 16 bits, indivisíveis, chamados SP (Stack Pointer) e PC (Program Counter). A função principal do SP é indicar o topo da pilha da memória RAM, usada principalmente para "salvar" valores dos registros e colocar endereços de retorno de sub-rotinas. O registro SP é afetado pelas instruções PUSH, POP, CALL e RET mas pode, entretanto, como os demais pares de registros internos, ser decrementado, incrementado, somado ou subtraído como HL, ser "salvo" num dado endereço da memória (e vice-versa) e ser carregado diretamente com o valor de HL ou com um dado de 2 bytes.

O registro PC é usado para as instruções de salto pois sempre indica qual a instrução que será executada. Vimos os saltos absolutos e os relativos, sendo que estes últimos tinham a vantagem de ter apenas 2 bytes e poder ser facilmente relocáveis. Foi então necessária a introdução aos números negativos para os saltos para trás, sendo que foi apresentada a instrução NEG que muda o sinal do número no acumulador.

Foram vistos também os saltos condicionais, que utilizam as flags (que estão no registro F) de CARRY, sinal (SIGN), PARTY/OVERFLOW e ZERO, notando que elas são atualizadas apenas quando é realizada alguma operação (apêndice 2). Vimos a instrução Compare (CP), para atualizar as flags quando quisermos, e as instruções de CALL e RET, que também podem ser usadas condicionalmente.

Vimos como fazer loops em Linguagem de Máquina usando a instrução DEC, que afeta a flag de ZERO, ou a instrução DJNZ, que usa o registro B como contador.

EXERCíCIOS

1- Faça um programa que execute um loop utilizando a instrução INC. Por exemplo, mande escrever na tela os primeiros 20 caracteres do seu MSX.
Observação: Você deverá usar a instrução CP para verificar o fim do loop.
Sugestão: Observe nas rotinas do BIOS se existe uma que imprima um caractere no vídeo.

2- Faça um programa que "varra" uma dada região de memória (por exemplo), os 60 bytes após o endereço 3D76H), coloque todos os números que forem menores que 66 (42H) na pilha (por exemplo, colocando os números no acumulador e fazendo PUSH AF) e, a seguir, consiga voltar ao BASIC sem problemas.

3- Faça uma sub-rotina que, por subtrações sucessivas, encontre o resultado inteiro da divisão de dois números positivos que estão no acumulador (dividendo) e registro E (divisor); o resultado deverá estar no registro D e nenhum outro registro deve ser alterado. A seguir, faça um programa que chame essa sub-rotina (veja o exemplo da multiplicação).

4- Faça um programa que coloque em ordem crescente o conteúdo dos registros B, C, D, E, H e L. Para verificar se seu programa funcionou, coloque os valores dos registros em bytes sucessivos (RAM) e elabore um programa em BASIC que faça um PEEK desses seis bytes.

5- Baseado no exercício 2 do capítulo 3 que, utilizando um programa em BASIC, passava parâmetros para uma sub-rotina em Linguagem de Máquina que fazia soma (ou subtração), repita o processo para a sub-rotina apresentada neste capítulo que faz a multiplicação. Em outras palavras, você terá um programa em BASIC que fornece dois números (a serem multiplicados) a uma sub-rotina em Linguagem de Máquina a qual retorna o resultado em BC.

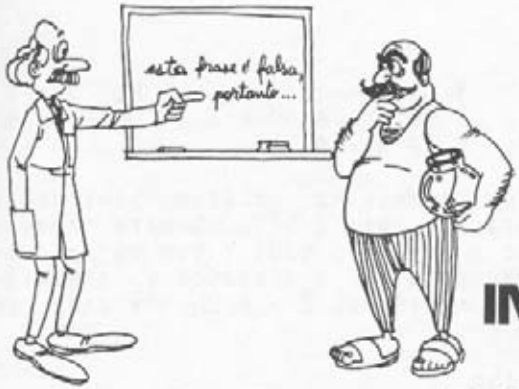
6- Faça um programa que subtraia (ou some) números com mais de 3 bytes, utilizando o registro B como contador de bytes e os registros C e A para efetuar as operações por partes. Os números devem estar nos bytes após o programa e sugerimos que você utilize os pares DE e HL como ponteiros para cada número. A seguir faça um programa em BASIC que permita passar ao programa os seguintes parâmetros:

a) número de bytes;
b) endereço inicial do primeiro número;
c) endereço inicial do segundo número;

7- Faça um programa que retorne como valor na forma inteira o endereço para o qual o SP aponta.

INSTRUÇÕES VISTAS NESTE CAPÍTULO.

PUSH	par de registros (incluindo AF)
POP	par de registros
LD	SP,HL
LD	SP,dado
LD	SP,(endereço)
LD	(endereço),SP
ADD	HL,SP
ADC	HL,SP
SBC	HL,SP
INC	SP
DEC	SP
JP	endereço
JP	(condições: C, NC, PO, PE, Z, NZ, M, P), endereço
JR	dado (-128 a +127)
JR	(condições: C, NC, Z, NZ), dado
JP	(HL)
NEG	(ou NEG A)
CP	dado (ou CP A, dado)
CP	(HL) (ou CP A, (HL))
CP	registro (ou CP A, registro)
DJNZ	dado
CALL	endereço
RET	endereço
CALL	condições: C, NC, PO, PE, Z, NZ, M, P
RET	condições: C, NC, PO, PE, Z, NZ, M, P



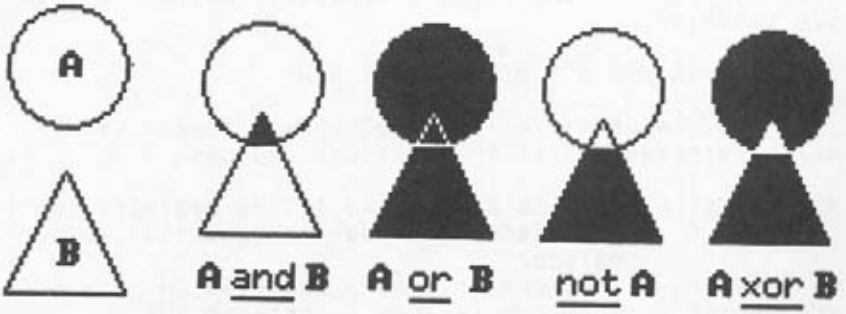
6

INSTRUÇÕES LÓGICAS E OPERAÇÕES COM BITS

OPERAÇÕES LÓGICAS

As instruções lógicas são instruções que mexem diretamente com os bits. Vamos inicialmente dar uma pequena explicação das principais operações lógicas que são: AND, OR, XOR e NOT. Para isso, basta associar aos bits a teoria dos conjuntos da seguinte maneira: ao número 1 associamos o conjunto universo e ao número 0, o conjunto vazio; a operação AND seria a operação de intersecção de conjuntos, a operação OR corresponderia à união e a função NOT ao complemento.

FIGURA 6.1 - Diagramas para operações AND, OR e NOT.



Logo:

```
1 AND 1 = 1              1 OR 1 = 1            NOT 0 = 1
1 AND 0 = 0 AND 1 = 0    1 OR 0 = 0 OR 1 = 1   NOT 1 = 0
0 AND 0 = 0              0 OR 0 = 0
```

Na Linguagem de Máquina, existem instruções que fazem essas operações bit a bit, sempre tendo o acumulador como sendo o registro onde ficam os resultados. Por exemplo, suponha que o conteúdo do acumulador seja A = 0C7H e do registro, B = 0A9H; falando em bits, temos:

```
A = 11000111B
B = 10101001B
```

assim:

```
A AND B = 10000001B = 81H
A OR B  = 11101111B = 0EFH
NOT A   = 00111000B = 38H
```

OBSERVAÇÃO: Colocamos sempre a letra B após os números binários.

Em outras palavras, a função AND só resulta 1 se os dois bits correspondentes forem 1, caso contrário, resulta 0; a função OR dá resultado 0 apenas se os dois bits correspondentes forem 0, senão, ela fornece o número 1; e a função NOT tem apenas um operando e coloca 0 onde havia 1 e vice-versa. Quanto à função XOR, ela fornece o resultado 1 somente se os bits correspondentes forem diferentes. Assim, no exemplo anterior:

```
A XOR B = 01101110B = 6EH
```

Seguem-se as instruções em Linguagem de Máquina correspondentes às operações lógicas:

AND registro: realiza a operação AND do registro com o acumulador deixando o resultado no acumulador.
OR registro : idem com relação à operação OR.
XOR registro: idem com relação à operação XOR.

que correspondem à figura 6.2.

FIGURA 6.2 - Instruções AND, OR e XOR.

INSTR.	CÓDIGO	INSTR.	CÓDIGO	INSTR.	CÓDIGO
AND A	A7H	OR A	B7H	XOR A	AFH
AND B	A0H	OR B	B0H	XOR B	A8H
AND C	A1H	OR C	B1H	XOR C	A9H
AND D	A2H	OR D	B2H	XOR D	AAH
AND E	A3H	OR E	B3H	XOR E	ABH
AND H	A4H	OR H	B4H	XOR H	ACH
AND L	A5H	OR L	B5H	XOR L	ADH

Essas três instruções afetam a flag do CARRY (deixando-a em 0) e todas as outras flags vistas até aqui; assim, uma maneira de se "zerar" o acumulador e, simultaneamente, a flag de CARRY é usar a instrução XOR A (ver apêndice 2). Como já deve ser esperado, o único registro que permite a operação NOT é o acumulador, com a seguinte instrução:

CPL (abreviação de ComPLement) (código 2FH)

que não altera a flag de CARRY nem as demais flags vistas até o presente momento.

Novamente poderia ser sugerida uma analogia para representar essas instruções; por exemplo:

```
AND A,A
OR A,C
XOR A,L
CPL A
```

Como o acumulador é sempre o alvo, ele é suprimido nas instruções "oficiais" (ou seja, aquelas interpretadas pelos proramas ASSEMBLERS), o mesmo ocorrendo para as instruções NEG, CP etc. Entretanto, poderia surgir a pergunta: então porque não fazer o mesmo com as instruções de soma e subtração? Bem, estamos também esperando essa resposta...

Daremos alguns exemplos para fixar melhor o funcionamento dessas instruções.

FIGURA 6.3 - Instrução AND.

```
C003 3EAF      LD    A,0AFH
C005 065A      LD    B,5AH
C007 A0        AND   B                  ;(ou AND A,B)
C008 6F        LD    L,A
C009 2600      LD    H,0
C00B 22F8F7    LD    (0F7F8H),HL
C00E 3E02      LD    A,2
C010 3263F6    LD    (0F663H),A
C013 C9        RET
```

temos:

A = 10101111B resultando A = 00001010B = 0AH = 10
B = 01011010B

FIGURA 6.4 - Instrução OR.

```
C003 3E82      LD    A,82H
C005 1E27      LD    E,27H
C007 B3        OR    E
C008 6F        LD    L,A
C009 2600      LD    H,0
C00B 22F8F7    LD    (0F7F8H),HL
C00E 3E02      LD    A,2
C010 3263F6    LD    (0F663H),A
C013 C9        RET
```

temos:

A = 10000010B resultando A = 10100111B = 0A7H = 167
E = 00100111B

FIGURA 6.5 - Instrução CPL (NOT) e XOR.

```
C003 3EF0      LD    A,0F0H
C005 2F        CPL                      ;(ou CPL A)
C006 6F        LD    L,A
C007 AF        XOR   A
C008 67        LD    H,A
C009 22F8F7    LD    (0F7F8H),HL
C00C 3E02      LD    A,2
C00E 3263F6    LD    (0F663H),A
C011 C9        RET
```

temos:

A = 11110000B resultando A = 00001111B = 0FH = 15

Execute estes programas modificando eventualmente os valores dos dados para fixar bem o funcionamento das operações lógicas.

Temos ainda as instruções que fazem essas operações utilizando o conteúdo da memória (endereçada por HL) ou um dado de 8 bits diretamente. Observe a figura 6.6.

FIGURA 6.6 - Instruções lógicas com a memória e dados de 1 byte.

INSTRUÇÃO	CÓDIGO
AND (HL)	A6H ;(ou AND, (HL))
OR (HL)	B6H ;(ou OR , (HL))
XOR (HL)	AEH ;(ou XOR, (HL))
AND dado	E6H +1 byte para o dado;(ou AND A,dado)
OR dado	F6H +1 byte para o dado;(ou OR A,dado)
XOR dado	EEH +1 byte para o dado;(ou XOR A,dado)

Como as instruções AND e OR afetam todas as flags, elas podem ser utilizadas para decidir se um dado número no acumulador é positivo, negativo ou zero quando queremos efetuar algum "salto", sem necessitar utilizar a instrução CP. Por exemplo, suponha que em dado ponto do programa você deseja "saltar" para o endereço C030H se o conteúdo do registro B for positivo:

FIGURA 6.7 - Uso da instrução OR para alterar apenas as FLAGS.

```
C003 78          LD    A,B          ;(ou OR A,A)
C004 B7          OR    A            ;não altera o acumu-
                                     lador, mas apenas
                                     as flags
C005 F230C0      JP    P,0C030H     ;salta para C030H
                                     se positivo
```

O mesmo poderia ter sido feito com AND A? E com CPL?

INSTRUÇÕES QUE AFETAM OS BITS

Vamos agora aprender instruções que modificam individualmente os valores dos bits dos registros ou do conteúdo da memória indicada por HL. Talvez no momento você não veja muita utilidade para essas instruções mas, se você se aprofundar na Linguagem de Máquina, verá como elas poderão ajudá-lo. Assim, temos as instruções de SET para colocar um determinado bit em 1 e RES (RESet) para colocá-lo em 0. Devido ao grande número de combinações possíveis, vamos montar duas tabelas (figuras 6.8 e 6.9).

FIGURA 6.8 - SET.

	A	B	C	D	E	H	L	(HL)
0	CBC7H	CBC0H	CBC1H	CBC2H	CBC3H	CBC4H	CBC5H	CBC6H
1	CBCFH	CBC8H	CBC9H	CBCAH	CBCBH	CBCCH	CBCDH	CBCEH
2	CBD7H	CBD0H	CBD1H	CBD2H	CBD3H	CBD4H	CBD5H	CBD6H
3	CBDFH	CBD8H	CBD9H	CBDAH	CBDBH	CBDCH	CBDDH	CBDEH
4	CBE7H	CBE0H	CBE1H	CBE2H	CBE3H	CBE4H	CBE5H	CBE6H
5	CBEFH	CBE8H	CBE9H	CBEAH	CBEBH	CBECH	CBEDH	CBEEH
6	CBF7H	CBF0H	CBF1H	CBF2H	CBF3H	CBF4H	CBF5H	CBF6H
7	CBFFH	CBF8H	CBF9H	CBFAH	CBFBH	CBFCH	CBFDH	CBFEH

FIGURA 6.9 - RESET.

	A	B	C	D	E	H	L	(HL)
0	CB87H	CB80H	CB81H	CB82H	CB83H	CB84H	CB85H	CB86H
1	CB8FH	CB88H	CB89H	CB8AH	CB8BH	CB8CH	CB8DH	CB8EH
2	CB97H	CB90H	CB91H	CB92H	CB93H	CB94H	CB95H	CB96H
3	CB9FH	CB98H	CB99H	CB9AH	CB9BH	CB9CH	CB9DH	CB9EH
4	CBA7H	CBA0H	CBA1H	CBA2H	CBA3H	CBA4H	CBA5H	CBA6H
5	CBAFH	CBA8H	CBA9H	CBAAH	CBABH	CBACH	CBADH	CBAEH
6	CBB7H	CBB0H	CBB1H	CBB2H	CBB3H	CBB4H	CBB5H	CBB6H
7	CBBFH	CBB8H	CBB9H	CBBAH	CBBBH	CBBCH	CBBDH	CBBEH

Note que os oito bits de cada registro são "numerados" de 0 a 7. Assim, por exemplo, se você quiser colocar no bit 5 do registro E o valor 1, deverá usar a instrução:

```
SET 5,E    0CBEBH
```

para colocar em 0 o bit 6 do byte indicado por HL:

```
RES 6,(HL)  CBB6H
```

Façamos um exemplo: vamos chamar uma sub-rotina que leia a coordenada X do 1o SPRITE na tabela de atributos, dado pelo endereço 6913 da VRAM. Se o parâmetro passado pelo BASIC for o número 1 na forma inteira, o bit 7 da coordenada X do SPRITE será setado. Caso contrário será ressetado.

```
        LD      HL,6913
        CALL    RDVRM
        LD      B,A
        LD      A,(0F7F9H)
        CP      0
        JR      Z,RESSETA
        SET     7,B
        JR      ESCRITA
RESSETA:
        RES     7,B
ESCRITA:
        LD      A,B
        CALL    WRTVRM
        RET
```

Se quisermos "testar" um determinado bit de um registro ou byte endereçado por HL para saber se ele é zero ou um, podermos usar a instrução BIT, que coloca o complemento no bit na flag Z do registro F. Assim, por exemplo, se você quiser "saltar" para um dado endereço (ou chamar uma sub-rotina), baseado na condição de que o bit 5 do registro E seja zero (0), basta fazer como apresentado na figura 6.10.

FIGURA 6.10 - A instução BIT.

```
BIT  5,E          ;coloca na flag CARRY o
                   complemento do bit 5 de E
JP   Z,endereço   ;(ou CALL Z,endereço)
```

(Para testar se o bit é 1, basta fazer JP NZ).

Se o bit 5 é 1, o complemento é 0 (colocado na flag Z); desse modo, a condição JP Z testa se a flag Z está em 1 (o que não ocorre) e o salto não é efetuado. Caso contrário (bit 5 em zero), o salto é efetuado.

Assim, temos as possíveis instruções na figura 6.11.

FIGURA 6.11 - Códigos da instrução BIT.

	A	B	C	D	E	H	L	(HL)
0	CB47H	CB40H	CB41H	CB42H	CB43H	CB44H	CB45H	CB46H
1	CB4FH	CB48H	CB49H	CB4AH	CB4BH	CB4CH	CB4DH	CB4EH
2	CB57H	CB50H	CB51H	CB52H	CB53H	CB54H	CB55H	CB56H
3	CB5FH	CB58H	CB59H	CB5AH	CB5BH	CB5CH	CB5DH	CB5EH
4	CB67H	CB60H	CB61H	CB62H	CB63H	CB64H	CB65H	CB66H
5	CB6FH	CB68H	CB69H	CB6AH	CB6BH	CB6CH	CB6DH	CB6EH
6	CB77H	CB70H	CB71H	CB72H	CB73H	CB74H	CB75H	CB76H
7	CB7FH	CB78H	CB79H	CB7AH	CB7BH	CB7CH	CB7DH	CB7EH

Temos também instruções para "setar" o bit de CARRY e complementá-lo; porém note que não há uma instrução para "re-setar" o CARRY.

FIGURA 6.12 - Set e Complement do CARRY.

INSTRUÇÃO	CÓDIGO	
SCF	37H	(Set Carry Flag)
CCF	3fH	(Complement Carry Flag)

Você saberia dizer uma instrução de 1 byte vista no capítulo anterior que seta a flag Z? Você saberia imaginar duas maneiras simples para colocar a flag de CARRY em zero sem alterar nenhum registro?

As instruções SET e RES não afetam nenhuma flag; as BIT afetam todas as flags menos o CARRY e as SCF e CCF afetam apenas o CARRY (considerando as flags vistas até o presente momento).

INSTRUÇÕES DE ROTAÇÃO

Veremos agora instruções que permitem fazer uma "rotação" dos bits dos registros ou memória indicada por HL. Algumas dessas instruções podem ser utilizadas para multiplicar ou dividir, facilmente, algum número por uma potência de 2 (2, 4, 8, 16, etc.).

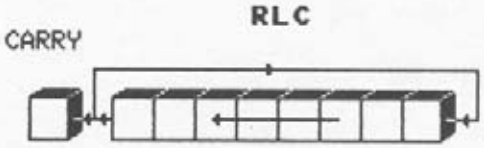
A primeira delas, chamada RLC, desloca todos os bits para a esquerda num movimento de "rotação", ou seja, o bit 7 é colocado no bit 0 e os bits de 0 a 6 são deslocados uma posição para a esquerda (o bit 7 é copiado no CARRY). Assim, por exemplo, se o registro D contiver:

10010010B e CARRY = 0

ao efetuarmos RLC D teremos:

D = 00100101B e CARRY = 1

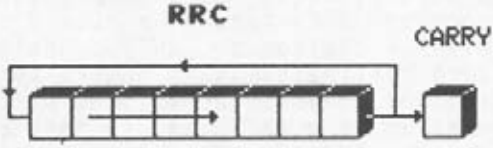
FIGURA 6.13 - RLC (Rotate Left with branch Carry).



A instrução RRC é análoga à RLC, efetuando a "rotação" para a direita e copiando o bit 0 no CARRY. Assim, se B = 11000011B e o CARRY = 1, ao efetuar RRC teremos:

B = 11100001B e CARRY = 1

FIGURA 6.14 - RRC (Rotate Rigth with branch Carry).

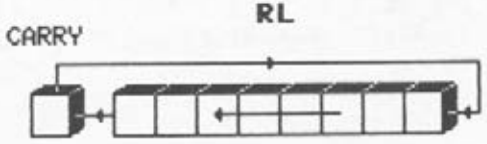


Temos também a instrução RL, que executa uma rotação para a esquerda como se o registro (ou byte apontado por HL) e o CARRY formassem um único registro

de 9 bits; ou seja, o CARRY é introduzido no bit 0, os bits de 0 e 6 são deslocados uma posição para a esquerda e o bit 7 é colocado no CARRY.Assim, se C = 01010101B e CARRY = 1. ao efetuar RLC teremos:

C = 10101011B e CARRY = 0

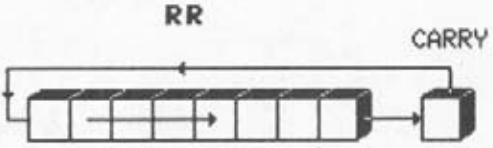
FIGURA 6.15 - RL (Rotate Left through carry).



A instrução RR é análoga à RL, fazendo uma "rotação" para a direita. Se A=00001111B e CARRY = 0. ao efetuar RR A teremos:

A = 00000111B e CARRY = 1

FIGURA 6.16 - RR (Rotate Right through carry).



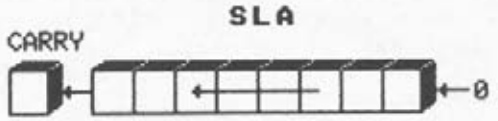
Vejamos agora a instrução SLA, que permite efetuar a multiplicação por alguma potência de 2; ela desloca todos os bits para a esquerda e "resseta" o bit 0 (o bit 7 é copiado no CARRY). Analogamente à base 10, quando multiplicamos um número por 10 simplesmente deslocamos todos os algarismos para a esquerda e acrescentamos um zero no final; na base 2, ao executar SLA estaremos multiplicando o número por 2, ao executá-la duas vezes sucessivas por 4, 3 vezes por 8 etc. Experimente a seguinte sub-rotina, colocada a partir do endereço C000H que multiplica por 2 um número inteiro menor que 255.

```
C003 2AF8F7    LD    HL,(0F7F8H)
C006 7E        LD    A,(HL)
C007 CB27      SLA   A
C009 77        LD    (HL),A
C00A 3E00      LD    A,0
C00C CE00      ADC   A,0
C00E 23        INC   HL
C00F 77        LD    (HL),A
C010 3E02      LD    A,2
C012 3263F6    LD    (0F663H),A
C015 C9        RET
```

O fato do bit ser colocado no CARRY nos permite detectar se houve um "estouro", ou seja, resultado maior que 255.

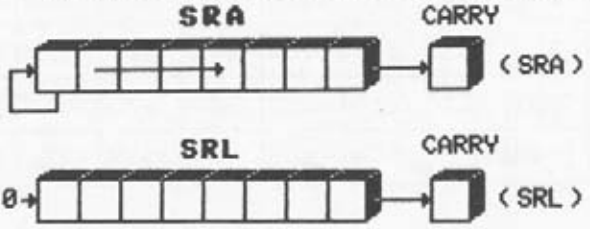
Quando houver a CARRY, ela será somada à parte MSB do parâmetro, que no caso deverá ser zero.

FIGURA 6.17 - SLA (Shift Left And clear least significant bit).



Por analogia, temos a instrução SRA, que desloca os bits para a direita, sem modificar os bits 7, e coloca o bit 0 no CARRY. Note, entretanto, que o fato do bit 7 ficar inalterado não permite dividir números positivos maiores que 127 (= 01111111B) mas possibilita a divisão de números negativos (>= -128). Entretanto, temos a instrução SRL, que desloca os 8 bits para a direita, copia o bit 0 no CARRY e resseta o bit 7. Com isto, podemos dividir números positivos até 255, mas não números negativos.

FIGURA 6.18 - SRA (Shift Right And preserve most significant bit) e SRL (Shift Right and clear most significant bit).



Veja o exemplo da figura 6.19 que divide um número da forma inteira menor que 255 por 2 (o número deve ser dado como parâmetro para a função USR).

FIGURA 6.19 - Divisão por 2.

```
C003 3AF8F7   LD    A,(0F7F8H)
C006 CB3F     SRL   A
C008 32F8F7   LD    (0F7F8H),A
C00B 3E00     LD    A,0
C00D 32F9F7   LD    (0F7F9H),A
C010 3E02     LD    A,2
C012 3263F6   LD    (0F663H),A
C015 C9       RET
```

Talvez não esteja muito clara a utilidade dessas instruções (principalmente a SRA); entretanto, como já havíamos dito, apenas o aprofundamento na Linguagem de Máquina poderá tornar as coisas mais claras (lembre-se que a estrutura da SRA permite a conservação do sinal na divisão). As instruções de rotação afetam todas as flags.

Segue uma tabela (figura 6.20) com as instruções de rotação:

FIGURA 6.20 - Instruções de rotação.

	REGISTROS							
INSTRUÇAO	A	B	C	D	E	H	L	(HL)
RLC	CB07H	CB00H	CB01H	CB02H	CB03H	CB04H	CB05H	CB06H
RRC	CB0FH	CB08H	CB09H	CB0AH	CB0BH	CB0CH	CB0DH	CB0EH
RL	CB17H	CB10H	CB11H	CB12H	CB13H	CB14H	CB15H	CB16H
RR	CB1FH	CB18H	CB19H	CB1AH	CB1BH	CB1CH	CB1DH	CB1EH
SLA	CB27H	CB20H	CB21H	CB22H	CB23H	CB24H	CB25H	CB26H
SRA	CB2FH	CB28H	CB29H	CB2AH	CB2BH	CB2CH	CB2DH	CB2EH
SRL	CB3FH	CB38H	CB39H	CB3AH	CB3BH	CB3CH	CB3DH	CB3EH

Apenas a título de complementação, apresentaremos as seguintes instruções:

FIGURA 6.21 - Instruções de rotação com apenas 1 byte.

INSTRUÇÃO	CÓDIGO
RLCA	07H
RRCA	0FH
RLA	17H
RRA	1FH

que produzem no acumulador o mesmo efeito que as instruções RLC A, RRC A, RL A e RR A, mas afetam apenas a flag de CARRY. Essas instruções existem no Z80 apenas para compatibilizá-lo com os microprocessadores 8080 e 8085 da INTEL.

RESUMO

Vimos neste capítulo as operações que trabalham com bits, começando pelas operações que têm analogia com a teoria de conjuntos, ou seja, AND (intersecção), OR (união) e NOT (complemento, feita pela instrução CPL); tivemos também a operação lógica XOR, que resulta 1 apenas se os bits correspondentes forem diferentes. Feito isso, vimos como "setar" e ressetar bits (SET e RES), inclusive o CARRY, e como testar bits (instrução BIT).

Finalizando, vimos as instruções de rotação, sendo que algumas nos permitem multiplicar e dividir registros por potências de 2.

EXERCÍCIOS

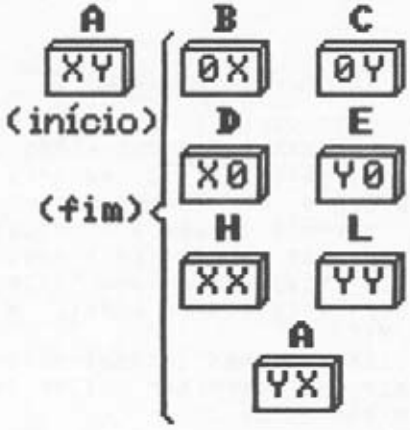
1- Usando as instruções vistas neste capítulo, faça uma sub-rotina capaz de de dividir o par de registros BC por 2. A seguir, faça um programa que chame duas vezes esta sub-rotina para dividir o conteúdo de BC por 4.

2- Faça uma programa que multiplique o registro C por 7 (sem usar somas sucessivas) e detecte se houve um "estouro" (testando o CARRY).

3- Uma boa aproximação para o número "pi" é 201/64. Faça um programa que, utilizando as sub-rotinas de multiplicação por somas sucessivas do capítulo anterior e a sub-rotina desenvolvida no exercício 1, multiplique o conteúdo do registro C por "pi", deixando o resultado em BC.

4- Faça um programa que forneça como saída o número de bits 1 de um dado registro; primeiramente, sem utilizar instruções de "rotação" e a seguir, utilizando-as.

5- Faça um programa que, dado o acumulador e chamando cada meio byte do mesmo de X e Y, respectivamente, forneça o seguinte:



6- Faça um programa que seja capaz de testar o bit 10 do SP.

7- Faça um programa que tire a média dos registros B e C (lembre-se que B + C pode ser maior que 255); a seguir, implemente o programa para que ele tire a média dos registros B, C, D e E. Use as instruções de rotação para fazer as divisões.

INSTRUÇÕES VISTAS NESTE CAPÍTULO

```
AND registro  (ou AND A, registro)
OR  registro  (ou OR  A, registro)
XOR registro  (ou XOR A, registro)
CPL           (ou CPL A)
AND (HL)   |
OR  (HL)   |
XOR (HL)   >  mesmas observações acima
AND dado   |
OR  dado   |
XOR dado   |
SET b,registro  |
RES b,registro  |
BIT b,registro  >  b: 0, 1, 2, 3, 4, 5, 6 ou 7
SET b,(HL)      |
RES b,(HL)      |
BIT b,(HL)      |
RLC registro
RRC registro
RL  registro
RR  registro
SLA registro
SRA registro
SRL registro
RLCA
RRCA
RLA
RRA
SCS
CCS
```

7

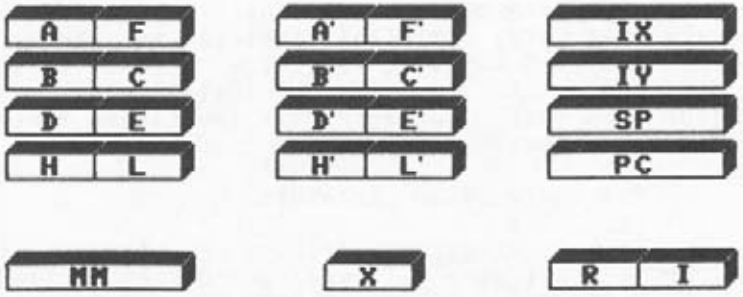
FINALIZANDO AS INSTRUÇÕES

Este capítulo encerra a apresentação das instruções disponíveis para se trabalhar em Linguagem de Máquina do microprocessador Z-80, fazendo uma pequena revisão dos conceitos gerais e introduzindo novas instruções.

REVISÃO DE CONCEITOS

Vamos inicialmente fazer um esquema indicando todos os registros internos do microprocessador, incluindo alguns novos que serão vistos neste capítulo:

FIGURA 7.1 - Como podemos imaginar os registros internos do Z-80.



Os registros A, B, C, D, E, H, L, F, A', B', C', D', E', H', L', F', I e R têm 8 bits cada e os registros SP, PC, IX e IY têm 16 bits cada. Os registros de 16 bits não podem ser divididos em dois registros de 8; por outro lado, alguns registros de 8 bits podem ser juntados para formar registros de 16 bits. Podemos então ter os pares: BC, DE, HL, B'C', D'E', e H'L' e, em alguns casos, AF e A'F'.

Como você deve lembrar, os registros internos permitem maior facilidade de operação e movimento que a memória (veja as instruções LD e aritméticas), sendo que o registro A (acumulador) é privilegiado por ser o alvo de todas as operações aritméticas e lógicas. O par HL também é privilegiado, comparado aos demais, com relação às instruções aritméticas e por permitir acessar a memória endereçada por ele (instruções que utilizam (HL)).

O registro PC serve para indicar onde, na memória, o microprocessador deve "buscar" informações; assim, por exemplo, ao executar em BASIC a função USR, o registro PC é carregado com o ponto de entrada da USR fazendo com que o microprocessador execute a sub-rotina em Linguagem de Máquina.

O registro F usa seus bits para dar informações a respeito de "estouros" matemáticos (CARRY e OVERFLOW), sinal do acumulador (SIGN,ZERO) e paridade (PARITY).

O registro SP é utilizado principalmente nas chamadas de sub-rotinas (para guardar o endereço de retorno na memória) e para "salvar" pares de registros (entretanto, usando a imaginação você poderá descobrir muitos outros usos para o SP). Assim, por exemplo, ao entrar em uma sub-rotina, é conveniente às vezes "salvar" todos os registros na entrada e obtê-los novamente na saída da sub-rotina; isto pode ser feito utilizando PUSH's e POP's. Esse procedimento coloca os registros nos bytes da pilha indicada por SP e, se quisermos "salvar" todos os registros, devemos fazer 4 PUSH's e 4 POP's (AF, BC, DE e HL). Entretanto, existe um método mais fácil, que economiza instruções e memória e usa a instrução:

EXX código 0D9H (Exchange)

Essa instrução utiliza os registros B', C', D', E', H' e L' (figura 7.1) para guardar os conteúdos de B, C, D, E, H e L de uma só vez; na realidade, ela troca os conteúdos dos registros um com o outro (B com

B', C com C' etc.). Para "salvar" os registros A e F, basta fazer:

```
EX AF,A'F'  código 08H (ou EX AF, AF')
```

que troca os conteúdos de A e F com A' e F'.

Existem mais duas instruções de trocas de registros que você poderá utilizar:

```
EX DE,HL  código 0EBH
```

que troca o conteúdo do registro D com H e do registro E com L, e

```
EX (SP),HL  código 0E3H
```

que troca o conteúdo da memória indicada pelo stack pointer (topo da pilha) com o registro L e o conteúdo da memória anterior com o registro H, sem alterar o valor de SP.

Note que os registros A', B', C', D', E', H', L' e F' só são acessíveis através das instruções EXX e EX. Naturalmente seu uso não se restringe apenas às sub-rotinas; basta deixar a imaginação "voar" e você logo descobrirá outras utilidades para eles.

Vamos agora analisar os registros IX e IY. Eles são registros de 16 bits cada que podem ser utilizados como o par de registros HL (só que não podem ser divididos em 2) para enderçar memórias e, além disso, permitem que uma constante seja somada ou subtraída ao endereço (desde que esta constante esteja entre -128 e +127). Assim, por exemplo, se o registro IX for carregado com D000H,

```
LD IX,0D000H
```

podemos fazer:

```
LD B,(IX + 40H)
```

que coloca no registro B o conteúdo do endereço D040H (note que isso não era possível para o par HL). Para saber qual o código em hexadecimal das instruções que utilizam IX e IY, imagine que você esteja utilizando o par HL; a seguir, coloque na frente do código resultante o byte 0DDH, se você utilizar IX, ou 0FDH, se utilizar IY; se IX ou IY estiver entre parênteses, então deve ser colocada uma constante, mesmo que ela

seja zero. Essa constante, que sempre ocupa 1 byte (-128 a +127) deve ser o terceiro byte do código, obrigatoriamente, mesmo que isso implique em dividir o código original em duas partes.

Vamos comparar o código de duas instruções na figura 7.2 .

FIGURA 7.2 - Comparação entre duas instruções.

Você conseguiria dizer qual o código de:

INC IX
INC (IY -20) (Utilize o apêndice 2 para ajudá-lo)?

COMPARAÇÃO DE BLOCOS

Você deve se lembrar como as flags são importantes para fazer desvios condicionais e que devemos estar sempre atentos sobre como cada instrução afeta as flags (para isto, basta usar o apêndice 2). Com relação a esse tipo de problema, a instrução CP é bastante útil! Esta instrução pode ser utilizada de maneira análoga às instruções LDI, LDD, LDIR, LDDR para comparar "alguma coisa" com uma dada região de memória, a fim de saber se esta "alguma coisa" está

nessa região. Assim temos as instruções da figura 7.3:

FIGURA 7.3 - Comparação de blocos.

INSTRUÇÃO	CÓDIGO
CPI	EDA1H (compara A com memória apontada por HL, INC HL e DEC BC)
CPD	EDA9H (idem, mas DEC HL)
CPIR	EDB1H (idem a CPI mas repete até BC ser zero)
CPDR	EDB9H (idem a CPD mas repete até BC ser zero)

que utilizam o par BC como contador, o par HL como ponteiro para a memória e o conteúdo do acumulador contendo o que nós desejamos comparar (ver capítulo 4). Assim, por exemplo, para saber se na região de memória entre D000H e D203H existe o número C3H, basta fazer como mostra a figura 7.4.

FIGURA 7.4 - Uso de CPIR.

```
LD    BC,0203H
LD    HL,0D000H
LD    A,0C3H
CPIR
```

Feito isto, basta verificar qual o conteúdo de HL para saber a posição em que está o número 0C3H. Obviamente, se houver mais que um byte com 0C3H nessa região, só a primeira será detectada; se não houver nenhum 0C3H, o conteúdo final de BC será zero (se isso ocorrer, a flag Z é colocada em zero; caso contrário, em um).

Vamos agora, a título de complementação, apresentar mais duas flags e mais algumas instruções, com o intuito de terminar todas as instruções disponíveis para Linguagem de Máquina do Z-80.

Comecemos pelas flags; lembre-se sempre que elas são modificadas só após efetuada alguma operação (ver apêndice 2):

- A flag de CARRY detecta "vai um" em adições e resultados negativos em subtrações, desde que se utilize a convenção de números positivos; serve também para ajudar as instruções de "rotação".
- A flag S (SIGN) indica o sinal do acumulador após

ter sido efetuada uma operação ou indica desigualdades (instrução CP).

- A flag de OVERFLOW detecta "estouros" quando fazemos operações utilizando números positivos e negativos, ajudando a detectar trocas de sinal indesejáveis.

- A flag de PARIDADE indica se o número de bits 1 (ou 0) do acumulador é par ou ímpar.

- A flag de ZERO, indica a igualdade do acumulador com determinado registro ou número (CP) ou se o acumulador ou dado registro (B, C, D, E, H e L) resultou zero, utilizando, por exemplo, DEC ou ADD. (Cuidado! Essa flag não funciona para pares de registros !

- A flag AC (Auxiliary Carry ou half CARRY) indica se houve um CARRY ("vai um") do bit 3 para o bit 4, ou seja, se houve um "estouro" ao somar a primeira metade do byte. Essa flag é utilizada internamente pelo microprocessador e não pode ser testada facilmente, um bom "truque" para testá-la, se necessário, seria fazer como o programa da figura 7.5 .

FIGURA 7.5 - Teste da flag AC.

```
PUSH AF        ;coloca AF no
                topo da pilha
POP  BC        ;coloca conteúdo
                do topo da pilha em BC
BIT  4,C       ;testa bit 4 do
                registro C
```

- A flag N (subtract flag) indica apenas se a última instrução executada foi uma soma ou subtração.

Assim, o registro F completo fica como apresentado na figura 7.6.

FIGURA 7.6 - Flags no registro F.

Note que os bits 5 e 3 não são utilizados para manter a compatibilidade com os microprocessadores 8080 e 8085.

Para complementar as instruções, comecemos por uma que não faz absolutamente nada.

NOP(código 00H)

NOP é abreviação de No OPeration. Seu efeito é análogo, por exemplo, a LD D,D.

É difícil explicar agora a utilidade desse tipo de instrução. A medida que seus programas se tornarem mais complexos você perceberá que instruções como NOP poderão ajudá-lo: por exemplo, suponha que no meio de dado programa você introduza vários NOP's em seguida. Naquela região é possível, mais tarde, substituir os NOP's por outras instruções, sem problema; mas, se não houverem os NOP's e você quiser acrescentar instruções no meio de programa, terá então que reescrever todo o programa a partir do ponto em que você quer a modificação. Outra utilização para o NOP é a retirada de instruções: basta substituir, na memória, a instrução por tantos NOP's quantos forem os bytes ocupados pela instrução, obviamente colocados a partir do endereço inicial da instrução. Suponha que você tivesse:

FIGURA 7.7 - Programa sem sentido.

```
C000 3E20      LD    A,32
C002 EB        EX    DE,HL
C003 DD3621A0 LD    (IX+21H),0A0H
C007 FD210080 LD    IY,8000H
C00B C9        RET
```

Para retirar a instrução LD (IX + 21H),0A0H 0DD3621A0H, por exemplo, basta fazer:

```
POKE  &HC003,0
POKE  &HC004,0
POKE  &HC005,0
POKE  &HC006,0
```

(Note que o código de NOP é 00H).

Nos exercícios 2 e 3 deste capítulo são sugeridos métodos para contornar o problema da modificação de programas em Linguagem de Máquina.

A seguir, temos a instrução RST (ReSTart), que tem o mesmo efeito que CALL mas ocupa apenas 1 byte. No entanto, ela é restrita, pois não possibilita

chamadas condicionais de sub-rotinas (assim,por exemplo, não podemos fazer RST NZ) e só permite acesso aos endereços 0, 8, 16 (10H), 24 (18H), 32 (20H), 40 (28H), 48 (30H) e 56 (38H). Assim, temos:

FIGURA 7.8 - A instrução RST.

INSTRUÇÃO	CÓDIGO
RST 0	0C7H (equivalente a CALL 0)
RST 8	0CFH (equivalente a CALL 8)
RST 10H	0D7H (equivalente a CALL 10H)
RST 18H	0DFH (equivalente a CALL 18H)
RST 20H	0E7H (equivalente a CALL 20H)
RST 28H	0EFH (equivalente a CALL 28H)
RST 30H	0F7H (equivalente a CALL 30H)
RST 38H	0FFH (equivalente a CALL 38H)

Nos MSX, fazer RST 0 produz um RESET no micro. Seu efeito é como se desligássemos o computador e tornássemos a ligá-lo em seguida.

Note que os endereços chamados pela instrução RST estão no começo da memória e, portanto, fazem parte da ROM. De fato alguns desses endereços são a entrada de sub-rotinas do BIOS, que poderão eventualmente ser utilizadas em outras programas.

A INTERRUPÇÃO

O Z-80 possue um pino chamado INT, e quando a tensão nesse pino atinge 0 volts o Z-80 entende que uma interrupção foi requisitada.

Terminando a execução da instrução, o PC passa a apontar para o endereço da próxima instrução e é salvo no STACK. Dependendo do modo de operação da interrupção, o Z-80 passará a executar uma sub-rotina em determinado endereço.

Os modos de operação da interrupção são 3, observe a figura 7.9.

Um desses modos deve ser escolhido no início do programa principal e pode ser mudado sempre que desejado.

FIGURA 7.9 - Os modos de Interrupção.

INSTRUÇÃO	CÓDIGO
IM 0	ED46
IM 1	ED56
IM 2	ED5E

No modo 0 o dispositivo externo, após interromper o microprocessador, é avisado pelo mesmo que a interrupção foi aceita (a instrução que estava sendo executada foi terminada e o PC foi colocado no topo da pilha). A seguir, o dispositivo deve fornecer ao Z-80 um código de um byte correspondente a uma instrução RST. Assim, no modo 0, poderemos ter sub-rotinas de interrupção que começam nos endereços 0000H, 0008H, 0010H, 0018H, 0020H, 0028H, 0030H ou 0038H, correspondentes aos oito possíveis códigos da instrução RST. Note bem que esta instrução não estará na memória do ROM. Os códigos de RST são fornecidos "via hardware". Note bem que esta instrução não está na memória do computador!

No modo 1 a interrupção faz com que o PC seja carregado com o endereço 0038H. Isto é equivalente ao modo 0 quando o dispositivo externo fornece o código de RST 38H. Entretanto, o modo 1 facilita o hardware, pois não é necessário que o dispositivo externo forneça o código. Em contrapartida, só temos um endereço possível e se tivermos mais do que um dispositivo que deverá interromper o microprocessador, não teremos como diferenciá-lo quando uma interrupção é requisitada.

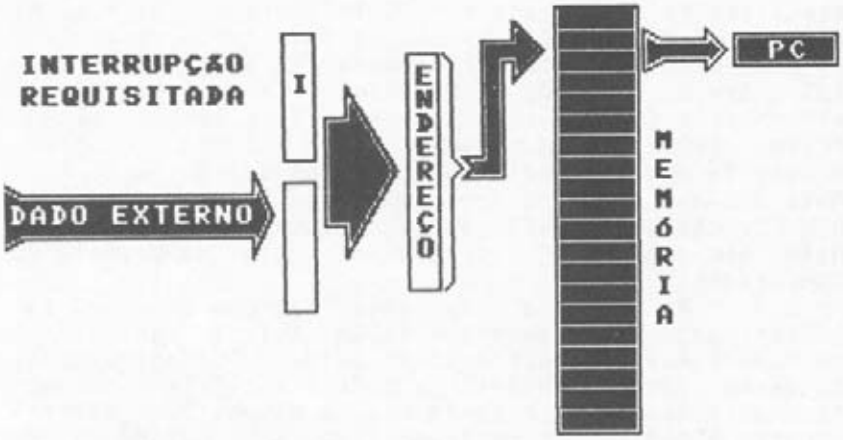
O modo 0 possibilita o uso de até 8 diferentes endereços para sub-rotinas de interrupção. Quando este número é insuficiente, utilizamos o modo 2. Este modo requer que sejam previamente colocados na RAM os endereços iniciais das sub-rotinas de interrupção, até um máximo de 128 endereços. Cada endereço tem dois bytes e, como sempre, o byte menos significativo deve ser colocado antes. Os endereços devem ser colocados em posições adjacentes formando uma tabela de endereços que pode ter até 256 bytes. O endereço inicial dessa tabela obrigatoriamente deve ter o byte menos significativo igual a zero.

Quando o Z-80 é interrompido no modo 2, o dispositivo externo deve fornecer um dado de um byte,

o qual é colocado juntamente com o conteúdo do registro I, formando um número de 16 bits (o registro I corresponde ao byte mais significativo) que deve apontar para alguma posição da tabela de endereços mencionada anteriomente. Os bytes correspondentes a essa posição e adjacentes são então colocados no PC. O Z-80 passa a executar, então, a sub-rotina que deve estar a partir do endereço dado por PC.

Veja na figura 7.10 como esse procedimento é feito.

FIGURA 7.10 - "Construção" do ponto de entrada no IM 2



O byte fornecido pelo dispositivo externo deve ser um número par. Mesmo que não seja, o Z-80 encarrega-se de zerar o bit 0. Você saberá explicar por quê?

O registro I pode ser carregado via programa e seu conteúdo pode ser colocado no acumulador:

FIGURA 7.11 - Instruções que utilizam o registro I.

INSTRUÇÃO	CÓDIGO
LD I,A	0ED47H
LD A,I	0ED57H

Assim, uma vez escolhido o modo de interrupção 2, os endereços iniciais das sub-rotinas de interrupção devem ser colocados numa região da RAM. O ende-

reço inicial dessa região deve ter 0 como byte menos significativo (você saberia agora explicar por quê?) e o byte mais significativo deve ser colocado no registro 0. Naturalmente o registro I pode ser modificado durante o programa. Você acha que isso é conveniente?

Como a interrupção pode acontecer em qualquer instante, podem existir ocasiões (geralmente partes importantes de um programa), em que não é conveniente que isto ocorra. Assim, existem duas instruções que permitem ou não que o Z-80 aceite interrupções. São elas:

FIGURA 7.12 - Instruções para permitir ou não interrupções.

INSTRUÇÃO	CÓDIGO	NOME E SIGNIFICADO
EI	0FBH	Enable Interrupt; permite a interrupção.
DI	0F3H	Disable Interrupt; não permite a interrupção

Desse modo, uma vez executada a instrução EI, o Z-80 aceitará interrupções até que seja encontrada uma instrução DI. A partir desse momento, mesmo que os dispositivos externos tentem interromper o Z-80, a requisição será ignorada. A situação inicial pode ser conseguida novamente com a instrução EI.

A sub-rotina de interrupção deve terminar com a instrução RETI (RETurn from Interrupt - código 0ED4DH) em vez de RET. Seu funcionamento é análogo ao RET só que, além das "funções normais" do RET, ela faz com que o hardware externo ao Z-80 saiba que a sub-rotina de interrupção está terminada. Note que se quisermos que as interrupções continuem habilitadas, devemos colocar uma instrução EI antes de RETI.

Nos MSX, o Z-80 está operando em IM 1, e é interrompido a cada 60 Hz pelo processador de vídeo (VDP).

Como o Z-80 está operando no modo 1, o endereço 38H é o ponto de entrada para a rotina que realiza os seguintes procedimentos:

- Incrementa a variável do sistema que fornece o TIME
- Verifica se existe alguma tecla pressionada e a coloca no buffer do teclado.

- Faz o caractere do cursor ser o inverso do caractere sobre o qual ele está (em SCREEN 0 e 1).
- Cuida das "QUEWEDS" do PLAY.

ENTRADA E SAÍDA DE DADOS (IN e OUT)

Para que o computador possa se comunicar com dispositivos externos (impressora, modem, vídeo, etc.) o Z-80 permite o acesso a 256 portas de I/O (0 - FFH).

Para enviar dados para um dispositivo é utilizada a instrução OUT. Veja na figura 7.13 as possíveis configurações do OUT.

FIGURA 7.13 - O comando OUT.

MNEMÔNICO	CÓDIGO
OUT (número),A	D3xxH
OUT (C),A	ED79H
OUT (C),B	ED41H
OUT (C),C	ED49H

MNEMÔNICO	CÓDIGO
OUT (C),D	ED51H
OUT (C),E	ED59H
OUT (C),H	ED61H
OUT (C),L	ED69H

Note que o registro entre parênteses deve conter o número da porta de I/O para o qual vai ser enviado o dado.

A instrução IN funciona da mesma forma, só que o dado é recebido do dispositivo em vez de ser enviado. Observe a figura 7.14 .

FIGURA 7.14 - A instrução IN.

MNEMÔNICO	CÓDIGO
IN A,(número)	DBxxH
IN A,(C)	ED78H
IN B,(C)	ED40H
IN C,(C)	ED48H

MNEMÔNICO	CÓDIGO
IN D,(C)	ED50H
IN E,(C)	ED58H
IN H,(C)	ED60H
IN L,(C)	ED68H

Para o envio e a recepção de blocos de dados, temos as instruções da figura 7.15.

FIGURA 7.15 - Instruções para envio e recepção de blocos de dados.

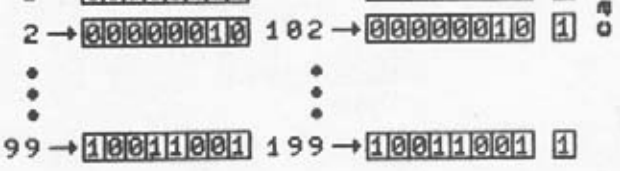
INSTRUÇÃO	CÓDIGO
INI	EDH A2H
IND	EDH AAH
INIR	EDH B2H
INDR	EDH BAH
OUTI	EDH A3H
OUTD	EDH ABH
OTIR	EDH B3H
OTDR	EDH BBH

O funcionamento dessas instruções é análogo às de deslocamento de blocos, só que o registro A indica qual o número da porta de I/O no qual serão enviados ou recebidos os dados.

OPERAÇÕES COM NÚMEROS DECIMAIS

O Z-80 possui instruções para facilitar, em certos casos, a utilização de números na base 10, de maneira a representar cada algarismo de (0 a 9) com meio byte ("0000 a "1001"). Assim temos a instrução DAA (Decimal Adjust Accumulator - código 27H) que "transforma" o conteúdo do acumulador de modo que cada meio byte (chamado também por "nibble") indique um dígito de 0 a 9 ("0000" a "1001" de tal maneira a representar em decimal o número que estava em hexadecimal (binário) no acumulador. Assim, com o auxílio do CARRY, poderemos representar números de 0 a 199.

FIGURA 7.16 - Números decimais de 0 a 199 com CARRY e acumulador.



Repare que devemos "olhar" para o acumulador como se ele contivesse um número hexadecimal. Naturalmente a instrução DAA não produzirá nunca os dígitos A, B, C, D, E e F.

Assim, por exemplo, se quisermos somar os números 16 e 26 (base.10) poderemos fazer:

16 + 26 = 42

16 = 0001 1100

1 6

26 = 0010 1100

2 6

FIGURA 7.17 - Uso da instrução DAA para somar números na base 10.

```
LD    A,16H
ADD   A,26H
DAA
```

Primeiramente note que colocamos os números como se fossem na base 16 (obviamente aqui não tem sentido utilizar os dígitos de A e F). Assim, após a instrução ADD A, 26, teremos A = 3CH (ou seja, 60 em decimal). A instrução DAA transforma o acumulador para 42H (= 16 + 26). Entretanto, o uso desta função é limitada, pois se tentarmos colocar este número na variável de retorno ao BASIC iremos obter 66 (por quê?). Tente imaginar como resolver este "problema". Repare que se tivéssemos somado 99H, após DAA iríamos ter A = 98H e CARRY = 1.

A maneira como esta instrução trabalha é razoavelmente complicada mas é importante perceber que o resultado depende do fato da instrução anterior ser uma soma ou subtração. Assim, se tivermos:

FIGURA 7.18 - Uso da instrução DAA para subtrair números na base 10.

```
LD    A,42H
SUB   06H
DAA
```

Após SUB A, 06H teremos, como no programa 1.1, A = 3CH. Entretanto, após DAA teremos A = 36H (42 - 6 = 36).

A instrução DAA afeta as flags de acordo com o resultado obtido.

Novamente salientamos que, para interfacear com o BASIC, esta instrução não é muito simples de ser utilizada. Entretanto, para aplicações "exclusivas" da Linguagem de Máquina ela pode ser muito útil...

O ASSEMBLY Z-80 apresenta duas instruções de rotação para utilizar números na base 10:

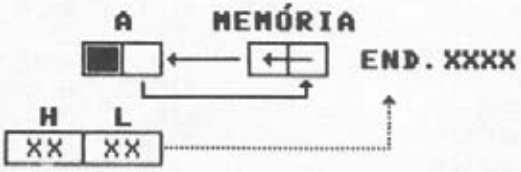
FIGURA 7.19 - Instruções de Rotação de nibbles.

INSTRUÇÃO	CÓDIGO	NOME COMPLETO
RLD	EDH 6FH	Rotate Left Decimal
RRD	EDH 67H	Rotate Right Decimal

Cuidado! Não as confunda com as instruções RL D (código CB12H) e RR D (código CB1AH) vistas anteriormente.

Elas fazem uma rotação de nibbles utilizando o byte indicado pelo par HL e o nibble menos significativo do acumulador. O nibble mais significativo do acumulador permanece inalterado. Assim, temos:

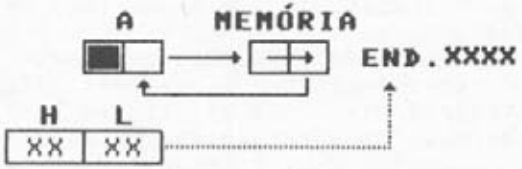
FIGURA 7.20 - Visualização da instrução RLD.



Se no microprocessador tivermos A = 02H e HL = C000H e, no endereço C000H, tivermos 19H, após RLD teremos A = 01H e no endereço C000H teremos 92H, ou seja, efetuamos uma rotação para a esquerda.

A instrução RRD é análoga, efetuando, entretanto, uma rotação para a direita.

FIGURA 7.21 - Visualização da instrução RRD.



Note que estas instruções implicam na rotação "simultânea" de quatro bits. Elas afetam todas as flags menos o CARRY. Sua utilização, assim como todas as instruções de rotação, depende muito da imaginação.

LABELS

Para finalizar o capítulo, apresentaremos o conceito de LABELS. Os LABELS são nomes que associamos às instruções e sub-rotinas em Linguagem de Máquina, ou melhor, a seus endereços, para facilitar a escrita do programa numa "primeira passada"; assim, todos os cálculos de endereços necessários para as instruções de salto ou chamadas de sub-rotinas podem ser feitos posteriormente. Observe o seguinte:

FIGURA 7.16 - Uso de LABELS.

```
C000 81        LOOP:  ADD   A,C
C001 0C               INC   C
C002 77               LD    (HL),A
C003 05               DEC   B
C004 20FA             JR    NZ,LOOP
C006 CD0CC0           CALL  NADA
C009 C300C0           JP    LOOP
C00C 00        NADA:  NOP
C00D 00               NOP
C00E C9               RET
```

Neste exemplo, LOOP e NADA são LABELS. Suponha que agora você queira colocar os códigos na memória e que o programa comece no endereço C000H. Os LABELS que estão à esquerda serão então calculados:

```
LOOP = C000H
NADA = C00CH
```

(veja que é importante saber quantos bytes tem cada instrução para se poder calcular os endereços que correspondem aos LABELS).

Nos saltos absolutos e chamadas sub-rotinas, os LABELS podem ser substituídos diretamente:

```
CALL  NADA - (CALL 0C000H código 0CD00C0H)
JP    LOOP - (JP   0C00CH código 0C30CC0H)
```

no entanto, nos saltos relativos, um pequeno cálculo é necessário:

```
JR NZ LOOP-(JR NZ -6 código 20FAH)-6 = 0FAH
```

Esse método facilita muito a escrita de um programa em Linguagem de Máquina e evita confusões desnecessárias pois, deixando todos os cálculos de endereços para o final, mantém sua atenção voltada para a lógica do programa sem "distraí-lo" com cálculos de endereços. Além disso, os LABELS (aliados aos comentários) facilitam a compreensão do programa (você logo perceberá isto...). Os programas ASSEMBLER'S, além de transformar automaticamente os mnemônicos em códigos, calculam os endereços relativos aos LABELS.

RESUMO

Fizemos uma rápida revisão dos conceitos vistos até este capítulo e introduzimos todos os registros internos do microprocessador, incluindo A', B', C', D', D', H', L, I, R, IX e IY.

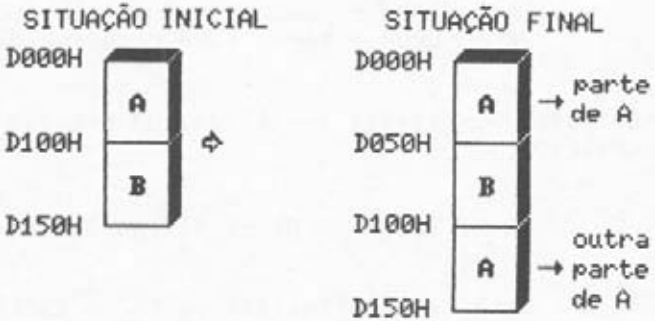
Terminamos de apresentar as instruções em Linguagem de Máquina, destacando as instruções de troca, as que utilizam IX e IY, e as comparações do acumulador com blocos de memória (CPI, CPIR, CPD, CPDR).

Finalmente, aprendemos o importante conceito dos LABELS, que facilita a escrita e posterior leitura do programa.

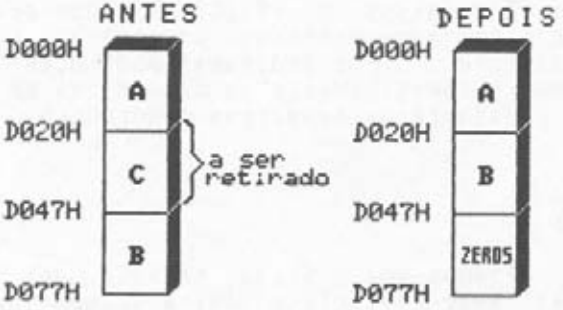
EXERCíCIOS

1- Você saberia explicar a diferença entre as instruções ADC A,A e RLA?

2- Escreva um programa, em Linguagem de Máquina, capaz de inserir um bloco de dados numa região de memória. Para verificar o funcionamento do programa, transfira um bloco B para dentro de um bloco A, inserindo-o a partir de D050H. Observe a figura a seguir. Utilize LABELS para facilitar a escrita do programa.



3- Faça agora um programa que "jogue fora" um bloco de dados e "encha" a memória livre com zeros, da seguinte maneira:



Você pode perceber a utilidade destes dois exercícios? O primeiro permite acrescentar instruções a um dado programa em Linguagem de Máquina enquanto o segundo permite retirar instruções, enchendo a memória que sobra no final com NOP's. Naturalmente, para torná-los práticos, seriam necessárias implementações, tais como um programa em BASIC que perguntasse "endereço para inserção?", "quantos bytes a serem inseridos" e "endereço inicial dos dados a serem inseridos?", para o ca-

so de ser necessário acrescentar instruções, e "endereço inicial de retirada?" e "quantos bytes a serem retirados?", para quando fosse preciso retirar instruções. Esse programa em BASIC deverá ser capaz de passar esses parâmetros para as sub-rotinas em Linguagem de Máquina.

4- Dada a lista de instruções a seguir, indique quais existem e quais não existem. No caso delas existirem, indique o número de bytes e as flags afetadas.

```
LD D,(BC)
LD A,(IX +16)
ADD HL,BC
EXX BC,DE
SUB HL,DE
LD (15095),A
LD (37040),22
AND (IX + 20H)
OR IY + 16
XOR HL
INC IX
LD B,(HL)
LD BC,(30000)
ADC HL,SP
ADD BC,SP
LD BC,DE
LD HL,SP
EXX SP,(HL)
CALL NC,31000
RST 16
LDDR
BIT1,(IX + 10)
SCF
NEG B
LD DE,14340
LD DE,(14340)
RL (HL)
RL HL
RR H
RES 0,(IY)
RES 0,IY
JR M,120
JR C,207
JP C,207
PUSH A' F'
POP BC
POP DC
```

```
PUSH L
RST 31
ADD A,261
SUB A,35
SUB C,D
ADC A,A
SUB E,E
SBC A,A
SBC HL,BC
SBC IX,DE
```

INSTRUÇÕES VISTAS NESTE CAPÍTULO

```
EXX
EX AF,A'F'
EX DE,HL
EX (SP),HL

Instruções usando:  IX - prefixo 0DDH
                    IY - prefixo 0FDH

CPI
CPD
CPIR
CPDR
RST M (M = 0, 8, 16, 24, 32, 40, 48 ou 56)
EI
DI
RETI
RETN
IM M (M = 0, 1 ou 2)
DAA
RLD
RRD

Instruções de entrada e saída: IN
                               OUT
```

8

INSTRUÇÕES SECRETAS DO Z-8Ø

Se você der uma olhada cuidadosa numa tabela de mnemônicos, perceberá que existem alguns "buracos" nas colunas "após CB" e "após ED". Talvez você já tenha se perguntado o que acontece se tentar utilizar essas "instruções". De fato, algo acontece e, por incrível que pareça, até com certa lógica! Temos, então, mais instruções disponíveis para utilizar o Z-80 que, no entanto, não são divulgadas em seus manuais (e portanto, não fazem parte das tabelas dos programas ASSEMBLER e DISASSEMBLER). Vamos, então, estudar essas "instruções secretas". O motivo pelo qual estas instruções não são divulgadas não é bem certo e preferimos não emitir opiniões tentando advinhar o por quê. O fato é que elas existem.

A ESTRUTURA DAS INSTRUÇÕES

Recordando conceitos vistos nos capítulos anteriores vimos que, se todas as combinações possíveis de um byte fossem utilizadas para indicar instruções, teríamos apenas 256 instruções disponíveis para Z-80. Assim, quatro bytes foram reservados para permitir o uso de mais instruções:

CBH e EDH que colocamos em "frente" a um dos 252 bytes restantes produzem outras instruções.

DDH e FDH que colocados em "frente" a quase todas as instruções que utilizam o par HL, permitem utilizar os pares IX e IY.

Desse modo, como oficialmente temos 248 instruções após CBH, 58 instruções após EDH e mais 140 instruções possibilitados por DDH e FDH temos:

252 + 248 + 58 + 140 = 698 instruções oficiais

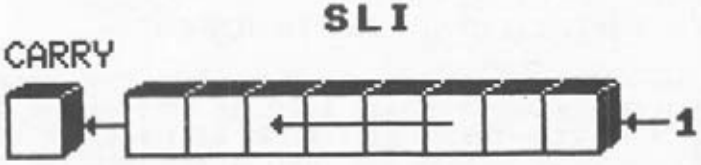
(que na realidade são "apenas" 696, pois os códigos 22H e ED63H correspondem ambos a LD (KK), HL e os códigos 2AH e ED68H correspondem ambos a LD HL, (KK).

Vamos agora tentar preencher os "buracos" restantes e fazer alguns malabarismos com os bytes para tentar "cavar" novas instruções. Num esforço de padronização, utilizaremos os mesmos mnemônicos apresentados na maioria da literatura a respeito dessas novas instruções.

AS INSTRUÇÕES NÃO-OFICIAIS

Comecemos então a preencher os oito bytes restantes para as instruções após CBH, ou seja, de 30H a 37H. Teremos então uma nova instrução de rotação similar à SLA só que em vez de resetar o bit 0, este é setado. Esta instrução é chamada de SLI (Shift Left Inverted) e corresponde a multiplicar o valor do registro ou memória por 2 e somar 1 ao resultado (o CARRY pode ser usado para detectar "estouros").

FIGURA 8.1 - A instrução SLI



Utilizando os bytes DDH e FDH temos então dez novas instruções:

FIGURA 8.2 - Dez "novas" instruções de rotação (-128(= Q (= 127)

INSTRUÇÃO	CÓDIGO
SLI A	CB37H
SLI B	CB30H
SLI C	CB31H
SLI D	CB32H
SLI E	CB33H
SLI H	CB34H
SLI L	CB35H
SLI (HL)	CB36H
SLI (IX+Q)	DDCBQ36H
SLI (IY+Q)	FDCBQ36H

Até agora, mencionamos que os bytes FDH e DDH poderiam ser colocados em frente de quase todas as instruções que envolvam o par HL para utilizar os registros IX e IY (de fato, as únicas "exceções" são as instruções EX DE,HL , EXX e as instruções que utilizam o par HL mas são precedidas pelo byte EDH). O que acontece se tentarmos utilizar FDH e DDH em frente de instruções que não utilizam o par HL? De fato, se colocarmos os bytes FDH e DDH em frente a qualquer instrução precedida por CBH, com exceção das instruções que utilizam o par HL e das instruções BIT, teremos 336 novas instruções que trabalham de modo bastante peculiar: elas executam a instrução no endereço indicado por IX e IY mais Q (número entre -128 e 127) e, após isto, copiam o resultado no registro indicado pela instrução! Note que isto equivale a duas instruções oficiais... Por exemplo:

```
SET 7,A = CBFFH        DDCBQFFH = SET 7,(IX+Q)
                                  LD  A,(IX+Q)
```

Fica difícil entretanto, definir um mnemônico para este tipo de instrução. Poderíamos, por exemplo, utilizar o seguinte:

```
SET 7,A/(IX+Q)
```

Aqui estão portanto essas 336 novas instruções:

FIGURA 8.3 - 112 "novas" instruções usando SET (-128(= @(=127).

reg/■ bit	A/(IX+@) A/(IY+@)	B/(IX+@) B/(IY+@)	C/(IX+@) C/(IY+@)	D/(IX+@) D/(IY+@)	E/(IX+@) E/(IY+@)	H/(IX+@) H/(IY+@)	L/(IX+@) L/(IY+@)
0	DDCB@C7H FDCB@C7H	DDCB@C0H FDCB@C0H	DDCB@C1H FDCB@C1H	DDCB@C2H FDCB@C2H	DDCB@C3H FDCB@C3H	DDCB@C4H FDCB@C4H	DDCB@C5H FDCB@C5H
1	DDCB@CFH FDCB@CFH	DDCB@C8H FDCB@C8H	DDCB@C9H FDCB@C9H	DDCB@CAH FDCB@CAH	DDCB@CBH FDCB@CBH	DDCB@CCH FDCB@CCH	DDCB@CDH FDCB@CDH
2	DDCB@D7H FDCB@D7H	DDCB@D0H FDCB@D0H	DDCB@D1H FDCB@D1H	DDCB@D2H FDCB@D2H	DDCB@D3H FDCB@D3H	DDCB@D4H FDCB@D4H	DDCB@D5H FDCB@D5H
3	DDCB@DFH FDCB@DFH	DDCB@D8H FDCB@D8H	DDCB@D9H FDCB@D9H	DDCB@DAH FDCB@DAH	DDCB@DBH FDCB@DBH	DDCB@DCH FDCB@DCH	DDCB@DDH FDCB@DDH
4	DDCB@E7H FDCB@E7H	DDCB@E0H FDCB@E0H	DDCB@E1H FDCB@E1H	DDCB@E2H FDCB@E2H	DDCB@E3H FDCB@E3H	DDCB@E4H FDCB@E4H	DDCB@E5H FDCB@E5H
5	DDCB@EFH FDCB@EFH	DDCB@E8H FDCB@E8H	DDCB@E9H FDCB@E9H	DDCB@EAH FDCB@EAH	DDCB@EBH FDCB@EBH	DDCB@ECH FDCB@ECH	DDCB@EDH FDCB@EDH
6	DDCB@F7H FDCB@F7H	DDCB@F0H FDCB@F0H	DDCB@F1H FDCB@F1H	DDCB@F2H FDCB@F2H	DDCB@F3H FDCB@F3H	DDCB@F4H FDCB@F4H	DDCB@F5H FDCB@F5H
7	DDCB@FFH FDCB@FFH	DDCB@F8H FDCB@F8H	DDCB@F9H FDCB@F9H	DDCB@FAH FDCB@FAH	DDCB@FBH FDCB@FBH	DDCB@FCH FDCB@FCH	DDCB@FDH FDCB@FDH

FIGURA 8.4 - 112 "novas" instruções usando RES (-128(= @(=127).

reg/■ bit	A/(IX+@) A/(IY+@)	B/(IX+@) B/(IY+@)	C/(IX+@) C/(IY+@)	D/(IX+@) D/(IY+@)	E/(IX+@) E/(IY+@)	H/(IX+@) H/(IY+@)	L/(IX+@) L/(IY+@)
0	DDCB@87H FDCB@87H	DDCB@80H FDCB@80H	DDCB@81H FDCB@81H	DDCB@82H FDCB@82H	DDCB@83H FDCB@83H	DDCB@84H FDCB@84H	DDCB@85H FDCB@85H
1	DDCB@8FH FDCB@8FH	DDCB@88H FDCB@88H	DDCB@89H FDCB@89H	DDCB@8AH FDCB@8AH	DDCB@8BH FDCB@8BH	DDCB@8CH FDCB@8CH	DDCB@8DH FDCB@8DH
2	DDCB@97H FDCB@97H	DDCB@90H FDCB@90H	DDCB@91H FDCB@91H	DDCB@92H FDCB@92H	DDCB@93H FDCB@93H	DDCB@94H FDCB@94H	DDCB@95H FDCB@95H
3	DDCB@9FH FDCB@9FH	DDCB@98H FDCB@98H	DDCB@99H FDCB@99H	DDCB@9AH FDCB@9AH	DDCB@9BH FDCB@9BH	DDCB@9CH FDCB@9CH	DDCB@9DH FDCB@9DH
4	DDCB@A7H FDCB@A7H	DDCB@A0H FDCB@A0H	DDCB@A1H FDCB@A1H	DDCB@A2H FDCB@A2H	DDCB@A3H FDCB@A3H	DDCB@A4H FDCB@A4H	DDCB@A5H FDCB@A5H
5	DDCB@AFH FDCB@AFH	DDCB@A8H FDCB@A8H	DDCB@A9H FDCB@A9H	DDCB@AAH FDCB@AAH	DDCB@ABH FDCB@ABH	DDCB@ACH FDCB@ACH	DDCB@ADH FDCB@ADH
6	DDCB@B7H FDCB@B7H	DDCB@B0H FDCB@B0H	DDCB@B1H FDCB@B1H	DDCB@B2H FDCB@B2H	DDCB@B3H FDCB@B3H	DDCB@B4H FDCB@B4H	DDCB@B5H FDCB@B5H
7	DDCB@BFH FDCB@BFH	DDCB@B8H FDCB@B8H	DDCB@B9H FDCB@B9H	DDCB@BAH FDCB@BAH	DDCB@BBH FDCB@BBH	DDCB@BCH FDCB@BCH	DDCB@BDH FDCB@BDH

FIGURA 8.5 - 112 "novas" instruções de rotação.

reg/m inst.	A/(IX+@) A/(IY+@)	B/(IX+@) B/(IY+@)	C/(IX+@) C/(IY+@)	D/(IX+@) D/(IY+@)	E/(IX+@) E/(IY+@)	H/(IX+@) H/(IY+@)	L/(IX+@) L/(IY+@)
RLC	DDCB@07H FDCB@07H	DDCB@00H FDCB@00H	DDCB@01H FDCB@01H	DDCB@02H FDCB@02H	DDCB@03H FDCB@03H	DDCB@04H FDCB@04H	DDCB@05H FDCB@05H
RRC	DDCB@0FH FDCB@0FH	DDCB@08H FDCB@08H	DDCB@09H FDCB@09H	DDCB@0AH FDCB@0AH	DDCB@0BH FDCB@0BH	DDCB@0CH FDCB@0CH	DDCB@0DH FDCB@0DH
RL	DDCB@17H FDCB@17H	DDCB@10H FDCB@10H	DDCB@11H FDCB@11H	DDCB@12H FDCB@12H	DDCB@13H FDCB@13H	DDCB@14H FDCB@14H	DDCB@15H FDCB@15H
RR	DDCB@1FH FDCB@1FH	DDCB@18H FDCB@18H	DDCB@19H FDCB@19H	DDCB@1AH FDCB@1AH	DDCB@1BH FDCB@1BH	DDCB@1CH FDCB@1CH	DDCB@1DH FDCB@1DH
SLA	DDCB@27H FDCB@27H	DDCB@20H FDCB@20H	DDCB@21H FDCB@21H	DDCB@22H FDCB@22H	DDCB@23H FDCB@23H	DDCB@24H FDCB@24H	DDCB@25H FDCB@25H
SRA	DDCB@2FH FDCB@2FH	DDCB@28H FDCB@28H	DDCB@29H FDCB@29H	DDCB@2AH FDCB@2AH	DDCB@2BH FDCB@2BH	DDCB@2CH FDCB@2CH	DDCB@2DH FDCB@2DH
SLI	DDCB@37H FDCB@37H	DDCB@30H FDCB@30H	DDCB@31H FDCB@31H	DDCB@32H FDCB@32H	DDCB@33H FDCB@33H	DDCB@34H FDCB@34H	DDCB@35H FDCB@35H
SRL	DDCB@3FH FDCB@3FH	DDCB@38H FDCB@38H	DDCB@39H FDCB@39H	DDCB@3AH FDCB@3AH	DDCB@3BH FDCB@3BH	DDCB@3CH FDCB@0CH	DDCB@3DH FDCB@3DH

Nos capítulos anteriores dissemos que os registros IX e IY (de 16 bits) não podiam ser divididos em 2 registros de 8 bits. Entretanto, se usarmos os bytes DDH e FDH em frente a qualquer instrução que utilize os registros H ou L separadamente (mas que não utilizem o par HL), executando as instruções precedi-

das por CBH ou EDH, teremos acesso às metades dos registros. Se utilizarmos instruções com o registro H, estamos lidando com o byte mais significativo de IX ou IY (que chamaremos HX E HY respectivamente) e, se utilizarmos instruções com o registro L, estaremos lidando com o byte menos significativo de IX ou IY (que chamaremos LX e LY respectivamente). Temos então 92 novas instruções:

FIGURA 8.6 - 52 "novas" instruções LD com XI e IY.

LD	HX	LX	HY	LY
A	DD7CH	DD7DH	FD7CH	FD7DH
B	DD44H	DD45H	FD44H	FD45H
C	DD4CH	DD4DH	FD4CH	FD4DH
D	DD54H	DD55H	FD54H	FD55H
E	DD5CH	DD5DH	FD5CH	FD5DH
HX	DD64H	DD65H		
LX	DD6CH	DD6DH		
HY			FD64H	FD65H
LY			FD6CH	FD6DH

A	B	C	D	E	K
DD67H	DD60H	DD61H	DD62H	DD63H	DD26KH
DD6FH	DD68H	DD69H	DD6AH	DD6BH	DD2EKH
FD67H	FD60H	FD61H	FD62H	FD63H	FD26KH
FD6FH	FD68H	FD69H	FD6AH	FD6BH	FD2EKH

Apesar do aspecto "estranho" da figura 8.6, ela é simples de utilizar. Os seguintes exemplos devem esclarecer a questão:

```
LD  C, HY =  FD 4CH
LD HX, LX =  DD 65H
LD LY, D  =  FD 6AH
LD HY, 16 =  FD 26 10H
```

Vamos agora às instruções aritméticas.

FIGURA 8.7 - 24 "novas" instruções aritméticas.

	HX	LX	HY	LY
ADD A,	DD84H	DD85H	FD84H	FD85H
ADC A,	DD8CH	DD8DH	FD8CH	FD8DH
INC	DD24H	DD2CH	FD24H	FD2CH
SUB A,	DD94H	DD95H	FD94H	FD95H
SBC A,	DD9CH	DD9DH	FD9CH	FD9DH
DEC	DD25H	DD2DH	FD25H	FD2DH

e, finalmente, as instruções lógicas e de comparação:

FIGURA 8.8 - 12 "novas" instruções lógicas e 4 de comparação.

	HX	LX	HY	LY
AND	DDA4H	DDA5H	FDA4H	FDA5H
OR	DDB4H	DDB5H	FDB4H	FDB5H
XOR	DDACH	DDADH	FDACH	FDADH
CP	DDBCH	DDBDH	FDBCH	FDBDH

Você pode perceber que ainda existem alguns "buracos" (nas instruções após EDH) e muitas outras

combinações possíveis. No entanto, até agora nada foi publicado a respeito dessas possíveis novas instruções, possivelmente porque elas produzem algum resultado "ilógico" ou resultado nenhum... Se você dispuser de tempo livre, "divirta-se" tentando descobrir o que estas instruções que faltam podem fazer... Entretanto, com estas 438 novas instruções temos agora 1136 instruções! Já é o suficiente para se divertir...

Você poderia perguntar sobre o efeito dessas novas instruções nas flags.. O efeito é equivalente às instruções "oficiais" e pode ser descoberto por analogia.

Novamente, salientamos que essas instruções não podem ser utilizadas com os ASSEMBLERs e DISASSEMBLERs. Entretanto, as pseudo instruções (ou NOPs posteriormente preenchidos) podem facilitar as coisas:

```
DB 0DDH
DEC L          ;equivalem à DEC LX

DB 0FDH
SBC A,H        ;equivalem à SBC A, HY
```

Observação: Alguns ASSEMBLERs utilizam duas pseudo instruções usadas para definir dados de um byte (Byte) ou dois bytes (Word), respectivamente:

DB e DW

Com relação aos registros HX, LX, HY e LY é conveniente salientar que sua utilização depende do seu programa: se você precisar de muitos registros e puder deixar de lado as facilidades de endereçamento de memória através dos pares IX e IY, então você pode utilizá-los sem problemas.

RESUMO

Neste "pequeno" capítulo apresentamos 438 novas instruções para o microprocessador Z-80 que, somadas às 698 instruções apresentadas previamente, perfazem um total de 1136 instruções!

Estas instruções foram "descobertas" utilizando os "buracos" existentes nas instruções após CBH e manipulando convenientemente os bytes DDH e FDH para

trabalhar com as metades dos registros IX e IY, "criando" assim, os registros HX, LX, HY e LY. Outras combinações são possíveis mas, aparentemente, não produzem instruções utilizáveis.

Salientamos o fato da não possibilidade do uso dessas "novas" instruções com os programas ASSEMBLERs e DISASSEMBLERs, o que pode ser compensado em parte para os ASSEMBLERs através de pseudo-instruções.

Comentamos também o fato de que a utilização dos registros HX, LX, HY e LY só é válida se não forem ser utilizadas as características de endereçamento de memória através de IX e IY.

EXERCÍCIO

1-) Procure modificar o programa MONITOR para que ele "funcione" com as instruções "secretas" apresentadas nesse capítulo.

9

PRIMEIRAS APLICAÇÕES

Nos oito capítulos anteriores vimos todas as instruções que o Z-80 dispõe. Neste capítulo veremos algumas "manhas" em Linguagem de Máquina no MSX. Como em outras linhas de microcomputadores, existem rotinas da ROM prontas para serem usadas ou detalhes do hardware que podem tornar dois programas que executem, auditiva ou visualmente a mesma tarefa, estruturalmente diferentes.

O BIOS

Ao ler o título acima você deve ter pensado:

- O BIOS!!! De novo!!! Será que esses caras não têm mais sobre o que falar? Muitos livros sobre MSX ficam horas e horas falando sobre ele.

A resposta é muito simples. Sim! Pois se você souber utilizar bem o BIOS, os seus programas em Linguagem de Máquina vão rodar corretamente em qualquer micro que respeite o padrão MSX (desde o micro do tio do primo do vizinho da sua irmã, até o MSX2 que aquele amigo chato "importabandeou" do Japão e no qual só sabe colocar cartuchos de joguinhos).

O BIOS (Sistema Básico de Entrada e Saída) é uma "mão na roda" para qualquer programador. Nele existem rotinas para o chaveamento de slots, ler e gravar bytes em outros slots, executar programas em

outros slots e se comunicar com o VDP, PPI, PSG, CASSETE, IMPRESSORA, etc.

Quando você tiver alguma idéia sobre um programa em Linguagem de Máquina, consulte uma relação das rotinas do BIOS. Certamente existirá uma que facilitará a execução do seu programa.

Vamos apresentar agora um programa que utiliza os 32K que ficam "escondidos" pelo interpretador BASIC.

Usaremos a página 0 do slot que possuir RAM para ficarmos "paginando" imagens gráficas, ou seja, trocando os valores do que estiverem na VRAM com os que estiverem no início da RAM.

A transferência será feita byte a byte seguindo a seguinte sequência.

- Lê-se um byte da VRAM;
- Lê-se um outro byte da página 0;
- Grava-se o outro byte na VRAM;
- Grava-se o byte na página 0;
- Verifica se terminou o Loop.

Para ler e gravar um byte na página 0 usaremos as rotinas WRSLT (0014H) e RDSLT (000CH) e na VRAM as rotinas WRTVDP (0047H) e RDVRM (004AH).

Como a rotina WRSLT pede como parâmetro o número do slot no qual será lido o byte utilizaremos o seguinte procedimento.

Nos MSX de 64 Kbytes de RAM reservam apenas um slot para essa memória . Ou seja, existe um slot que é preenchido totalmente com memória RAM.

Sabemos também que a página 3 que está habilitada é a do slot que contém a RAM pois caso contrário não teríamos as variáveis de sistema (que podem ser alteradas). Ao lermos a porta A da PPI obtemos o número dos slots que estão sendo acessados. Os bits 6 e 7 (página 3) indicarão o slot que contém a RAM. Rode o programa da figura 9.1 com o seguinte programa em BASIC.

```
10 DEFUSR = &HC000 : SCREEN 2
20 FOR L = 0 TO 100 STEP 3
30 CIRCLE (128,96),L : NEXT
40 A = USR(0)
50 SCREEN 2
60 FOR L = 0 TO 20
70 LINE (255*RND(1),192*RND(1))-(255*RND
(1),192*RND(1)),,B : NEXT
80 A=USR(0) : GOTO 80
```

FIGURA 9.1 - Programa de paginação.

```
C000 DBA8          IN   A,(PORTAA)
C002 CB07          RLC  A
C004 CB07          RLC  A
C006 E603          AND  00000011B
C008 322EC0        LD   (SLOT),A
C00B 210000        LD   HL,0000H
C00E          TRNSF:
C00E CD4A00        CALL RDVRM     ;lê 1 byte da VRAM
C011 322DC0        LD   (VBYTE),A ;guarda A em VBYTE
C014 3A2EC0        LD   A,(SLOT)  ;lê 1 byte da
C017 CD0C00        CALL RDSLT     ;página 0
C01A CD4D00        CALL WRTVRM    ;escreve A na VRAM
C01D 3A2DC0        LD   A,(VBYTE) ;
C020 5F            LD   E,A       ;escreve VBYTE
C021 3A2EC0        LD   A,(SLOT)  ;na página 0
C024 CD1400        CALL WRSLT     ;
C027 23            INC  HL        ;incrementa HL
C028 CB74          BIT  6,H       ;se o bit 6 de
C02A 28E2          JR   Z,TRNSF   ;H for 1, retorna
C02C C9            RET
C02D 00      VBYTE: DEFB 0
C02E 02      SLOT:  DEFB 02
```

Vamos ver agora um programa que: Dada uma string como parâmetro, a escreve na tela gráfica com espaçamento de SCREEN 0.

Para conseguirmos tal efeito utilizaremos duas variáveis de sistema GRPACX (FCB7H) e GRPACY (FCB9H) que indicam a coordenada a partir da qual ser impresso o caractere.

Para imprimir um caractere na tela gráfica, usaremos a rotina GRPPRT (008DH) que pede como parâmetro o código do caractere a ser impresso. Esse código deve ser colocado no Acumulador.

A GRPPRT incrementa automaticamente em 8 a variável do sistema GRPACX (FCB7H), então para que possamos imprimir com o espaçamento de SCREEN 0 devemos, após chamar essa rotina subtrair 2 da variável do sis- tema GRPACX.

Vejamos então o programa na figura 9.2.

FIGURA 9.2 - Programa de impressão na tela gráfica.

```
C000 CD7200        CALL INIGRP
C003 210000        LD   HL,0000H
C006 22B7FC        LD   (GRPACX),HL
C009 22B9FC        LD   (GRPACY),HL
C00C 2AF8F7        LD   HL,(0F7F8H)
C00F 46            LD   B,(HL)
C010 23            INC  HL
C011 5E            LD   E,(HL)
C012 23            INC  HL
C013 56            LD   D,(HL)
C014 1A      LOOP: LD   A,(DE)
C015 CD8D00        CALL GRPPRT
C018 13            INC  DE
C019 3AB7FC        LD   A,(GRPACX)
C01C D602          SUB  02H
C01E 32B7FC        LD   (GRPACX),A
C021 10F1          DJNZ LOOP
C023 CD9F00        CALL 09FH
C026 C9            RET
```

Podemos, também, fazer algumas brincadeiras interessantes com a impressora.

O programa a seguir (figura 9.3) transforma o seu MSX em uma máquina de escrever.

Usaremos a rotina CHGET (009FH) para esperarmos o pressionamento de uma tecla, e a rotina OUTLPT (014DH) para enviar o código ASCII da tecla pressionada para a impressora.

Para sair do programa, usaremos a rotina BREAKX (00B7H) que testa o pressionamento das teias CTRL + STOP.

FIGURA 9.3 - MSX como máquina de escrever.

```
C000 CD9F00    INICIO:CALL  CHGET
C003 CD4001           CALL  OUTDLP
C006 CDB700           CALL  BREAKX
C009 30F5             JR    NC,INICIO
C00B C9               RET
```

Com o programa desse jeito, as teclas que você pressiona não aparecem no vídeo, mas se usarmos a rotina CHGET (009FH) do BIOS poderemos ver o que escrevemos.

Altere o programa para que ele fique como mostra a figura 9.4.

FIGURA 9.4 - MSX como máquina de escrever e imprimindo no vídeo.

```
C000 CD9F00    INICIO:CALL  CHGET
C003 CDA200           CALL  CHPUT
C006 CD4001           CALL  OUTDLP
C009 CDB700           CALL  BREAKX
C00C 30F2             JR    NC,INICIO
C00E C9               RET
```

Note que esse programa foi feito usando-se quase só rotinas do BIOS!

O nosso próximo programa continuará trabalhando com a impressora.

Até o lançamento desse livro não havia no mercado nenhuma impressora que imprimisse todos os caracteres do MSX. Mas a maioria das impressoras possuem o recurso do modo gráfico, que nos dá o controle sobre as agulhas. Então por que não imprimirmos todos os caracteres em modo gráfico.

Na tabela de caracteres, eles estão armazenados como mostra a figura 9.5.

Se enviarmos essa sequência de bytes para a impressora trabalhando em modo gráfico obteremos a fi-

gura 9.6 .

FIGURA 9.5 - Matriz de um caractere.

1BC7H	0	0	1	1	1	1	0	0
1BC8H	0	1	0	0	0	0	1	0
1BC9H	1	0	1	0	0	1	0	1
1BCAH	1	0	0	0	0	0	0	1
1BCBH	1	0	1	0	0	1	0	1
1BCCH	1	0	0	1	1	0	0	1
1BCDH	0	1	0	0	0	0	1	0
1BCEH	0	0	1	1	1	1	0	0

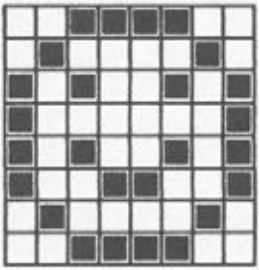
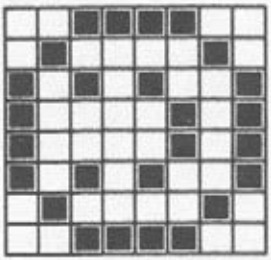


FIGURA 9.6 - Caractere "deitado".



A letra aparece deitada, porque no vídeo os bytes são transformados em segmentos horizontais e na impressora os bytes são transformados em segmentos verticais.

Vejamos como poderemos fazer para imprimí-los corretamente.

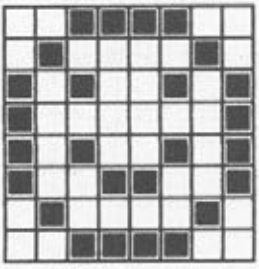
Se ao lermos os 8 bytes que formam o desenho do caractere construirmos uma tabela de 8 bytes formados pelos bits de mesmo número, conseguiremos imprimir o caractere corretamente na impressora.

Observe na figura 9.7 este processo.

FIGURA 9.7 - Impressão correta de um caractere.

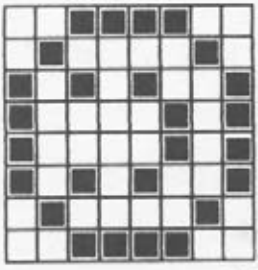
TABELA ORIGINAL

1BC7H
1BC8H
1BC9H
1BCAH
1BCBH
1BCCH
1BCDH
1BCEH



C200H
C201H
C202H
C203H
C204H
C205H
C206H
C207H

TABELA RODADA



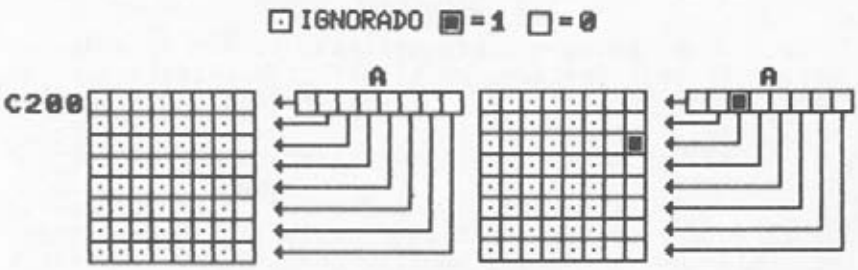
Montaremos, na figura 9.8, uma sub-rotina que, dado o endereço inicial dos 8 bytes do caractere em DE monta uma tabela com os bytes "verticalizados" a partir do endereço C200H.

FIGURA 9.8 - Rotina de "rotação" de um caractere.

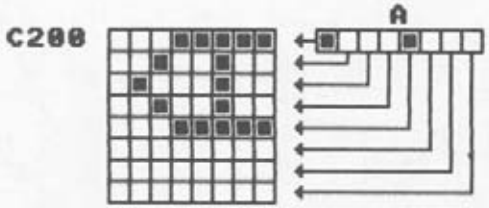
```
C000            VERTIC:
C000 0608           LD    B,08H       ;contador de bytes
C002            LOOPBY:               ;
C002 C5             PUSH  BC          ;preserva contador
C003 2100C2         LD    HL,0C200H   ;aponta HL p/ tabela
C006 0608           LD    B,08H       ;contador de bits
C008 1A             LD    A,(DE)      ;da matriz
C009            LOOPIT:               ;
C009 17             RLA               ;insere bit 7 na CARRY
C00A CB16           RL    (HL)        ;insere CARRY na tabela
C00C 23             INC   HL          ;incrementa elemento
C00D 10FA           DJNZ  LOOPIT      ;da tabela
C00F C1             POP   BC          ;efetua loop
C010 13             INC   DE          ;recupera contador
C011 10EF           DJNZ  LOOPBY      ;incrementa apontador
                                      ;efetua loop
C013 C9             RET               ;retorna
```

O LOOPBY carrega sucessivamente o acumulador com o valor de uma linha e o LOOPIT coloca cada bit do acumulador no bit 0 de um byte da tabela. Observe a figura 9.9.

FIGURA 9.9 - Visualização do programa.



e assim sucessivamente, eté o fim da rotina, quando a tabela está assim:



Note que essa rotina tem muitas aplicações como "rodar" padrões de caracteres ou SPRITES 8x8.

Usaremos, no próximo programa, a rotina da figura 9.8 para imprimirmos uma STRING em modo gráfico mas fazendo as devidas correções.

FIGURA 9.10 - Impressão de STRING em modo gráfico.

```
C000 2AF8F7          LD    HL,(0F7F8H) ;aponta HL para o
                                        descritor da string
C003 46              LD    B,(HL)      ;carrega B com com-
                                        primento da string
C004 CD3AC0          CALL  MGRA        ;coloca impressora
                                        em modo gráfico
C007 23              INC   HL          ;aponta HL para
C008 5E              LD    E,(HL)      ;o 1º caractere
C009 23              INC   HL          ;da string
C00A 56              LD    D,(HL)      ;
C00B EB              EX    DE,HL       ;
C00C         LOOPCA: ;
C00C E5              PUSH  HL          ;preservaapontador
C00D 7E              LD    A,(HL)      ;lê caractere
C00E C5              PUSH  BC          ;
C00F         CALCULO: ;
C00F 21BF1B          LD    HL, 1BBFH   ;calcula endereço
C012 5F              LD    E,A         ;no qual se inicia
C013 1600            LD    D,0         ;a matriz do caractere
C015 CB23            SLA   E           ;
C017 CB12            RL    D           ;
C019 CB23            SLA   E           ;
C01B CB12            RL    D           ;
C01D CB23            SLA   E           ;
C01F CB12            RL    D           ;
C021 AF              XOR   A           ;
C022 ED5A            ADC   HL,DE       ;
C024 EB              EX    DE,HL       ;
C025 CD5CC0          CALL  VERTIC      ;roda matriz do caractere
C028 0608            LD    B,8         ;
C02A 2100C2          LD    HL,0C200H   ;
C02D         LOOP0:  ;
C02D 7E              LD    A,(HL)      ;imprime matriz
C02E CDA500          CALL  LPTOUT      ;do caractere
C031 23              INC   HL          ;
C032 10F9            DJNZ  LOOP0       ;
C034 C1              POP   BC          ;
C035 E1              POP   HL          ;
C036 23              INC   HL          ;
C037 10D3            DJNZ  LOOPCA      ;
C039         FIM:    ;
C039 C9              RET               ;
C03A         MGRA:   ;
C03A 3E1B            LD    A,1BH       ;envia ESC
C03C CDA500          CALL  LPTOUT      ;
C03F 3E4B            LD    A,'K'       ;envia 'K'
C041 CDA500          CALL  LPTOUT      ;
C044 58              LD    E,B         ;envia nº de
C045 1600            LD    D,0         ;caracteres da
C047 CB23            SLA   E           ;string x 8
C049 CB12            RL    D           ;
C04B CB23            SLA   E           ;
C04D CB12            RL    D           ;
C04F CB23            SLA   E           ;
C051 CB12            RL    D           ;
C053 78              LD    A,E         ;
C054 CDA500          CALL  LPTOUT      ;
C057 7A              LD    A,D         ;
C058 CDA500          CALL  LPTOUT      ;
C05B C9              RET               ;
C05C         VERTIC: ;
C05C 0608            LD    B,08H       ;
C05E         LOOPBY: ;
C05E C5              PUSH  BC          ;rotina que roda
C05F 2100C2          LD    HL,0C200H   ;a matriz de
C062 0608            LD    B,08H       ;caracteres
C064 1A              LD    A,(DE)      ;
C065         LOOPIT: ;
C065 17              RLA               ;
C066 CB16            RL    (HL)        ;
C068 23              INC   HL          ;
C069 10FA            DJNZ  LOOPIT      ;
C06B C1              POP   BC          ;
C06C 13              INC   DE          ;
C06D 10EF            DJNZ  LOOPBY      ;
C06F C9              RET               ;
C070                 END               ;
```

No próximo programa trabalharemos um pouco com o processador de som. Faremos um programa que manipula os 14 registradores que controlam as frequências, volumes, envoltórias e mixagens de ruído.

Existe uma rotina do BIOS chamada WRPSG (WRite into Programable Sound Generator) que pede o número do registro do PSG no registro A e o valor no registro E.

O programa constitue-se de 7 blocos. O principal é chamado de LOOPRI (LOOP PRincipal). É nele que é feita a leitura do teclado e de acordo com a tecla pressionada é efetuado o salto para uma outra rotina.

A segunda rotina mais importante é a de envio dos valores dos 13 registros do PSG que se inicia a partir do "label" ENVIO. Ela lê o conteúdo da tabela TABREG (TABela de REGistros), imprime o número do registro no vídeo e o seu valor; além de enviar esse valor para o PSG. A impressão dos valores em hexadecimal no vídeo é feita pela rotina IMPHEX (IMPRIME em HEXadecimal).

Existem ainda as rotina AUMENTA e DIMINUE que alteram o valor de um dos elementos da tabela que é dado pela váriavel REGOP (REGistro OPerante). Esta variável é alterada pelo bloco iniciado pelo label MUDREG. E ,finalmente, existe o bloco OUTMIX que facilita a programação do registro 7 no qual os bits indicam se em um determinado canal há som, ruído ou os dois. Esta rotina espera o pressionamento de 8 teclas. as diferente de 0 "setarão" um determinado bit e as iguais a zero o "resetarão".

As teclas de seta (esquerda ou direita) alteram o valor de um registro e a tecla para cima passa o controle para o próximo registro.

FIGURA 9.11- Programa PROSOM (PROgramador de SOM).

```
C000 CDC300          CALL  CLS          ;limpa o vídeo
C003 213CC1          LD    HL,TABNOM    ;imprime mensagem no vídeo
C006 110000          LD    DE,0000H     ;
C009 010A00          LD    BC,000AH     ;
C00C CD5C00          CALL  LDIRVM       ;
C00F 3E07            LD    A,7          ;
C011 1E38            LD    E,00111000B  ;
C013 CD9300          CALL  WRTPSG       ;
C016 3E08            LD    A,8          ;
C018 1E0F            LD    E,15         ;
C01A CD9300          CALL  WRTPSG       ;
C01D 3E09            LD    A,9          ;
C01F CD9300          CALL  WRTPSG       ;
C022 3E0A            LD    A,10         ;
C024 CD9300          CALL  WRTPSG       ;
C027                 LOOPRI:            ;
C027 2601            LD    H,1          ;posiciona cursor na
C029 2E02            LD    L,2          ;linha 2 e coluna 1
C02B CDC600          CALL  POSIT        ;
C02E CD76C0          CALL  ENVIO        ;chama rotina de envio
C031 CD9F00          CALL  CHGET        ;espera tecla ser pressionada
C034 FE5A            CP    'Z'          ;compara com a letra Z
C036 C8              RET   Z            ;se igual, retorna ao BASIC
C037 FE1C            CP    28           ;compara com a seta à direita
```

```
C039 280F               JR    Z.AUMENTA     ;se igual, salta para a
                                             rotina aumenta
C03B FE1D               CP    29            ;compara com a seta à esquerda
C03D 2821               JR    Z.DIMINU      ;se igual, salta para a
                                             rotina diminui
C03F FE1E               CP    30            ;compara com a seta para cima
C041 2865               JR    Z,MUDCA       ;se igual, salta para a
                                             a rotina MUDCA
C043 FE20               CP    ' '           ;compara com a barra de espaço
C045 CAC8C0             JP    Z.ALTMIX      ;se igual, salta para ALTMIX
C048 18DD               JR    LOOPRI        ;salta para LOOPRI
C04A             AUMENTA:                   ;
C04A DD2110C1           LD    IX.REGISTROS;aponta IX para registros
C04E 3A2AC1             LD    A, (REGOP)    ;carrega A com o REGOP
C051 1600               LD    D,0           ;e o insere em DE
C053 5F                 LD    E.A           ;
C054 DD19               ADD   IX.DE         ;aponta IX para o elemento
                                             da tabela a ser alterado
C056 DD7E00             LD    A,(IX)        ;carrega A com o conteúdo
C059 C601               ADD   A.1           ;do registro e adiciona 1
C05B DD7700             LD    (IX),A        ;atualiza novo valor na tabela
C05E 18C7               JR    LOOPRI        ;retorna ao LOOPRI
C060             DIMINU:                    ;
C060 DD2110C1           LD    IX.REGISTROS;aponta IX para REGISTROS
C064 3A2AC1             LD    A, (REGOP)    ;carrega A com REGOP
C067 1600               LD    D.0           ;insere o nº do registro em DE
C069 5F                 LD    E.A
C06A DD19               ADD   IX,DE         ;aponta IX para o elemento
                                             da tabela a ser alterado
C06C DD7E00             LD    A,(IX)        ;carrega A com o conteúdo
C06F D601               SUB   1             ;do registro e subtra: 1
C071 DD7700             LD    (IX),A        ;atualiza valor na tabela
C074 1881               JR    LOOPRI        ;salta para LOOPRI
C076             ENVIO:                     ;
C076 3A28C1             LD    A,(MISTUR)    ;carrega A com MISTUR
C079 DD2110C1           LD    IX ,REGISTROS;aponta IX para   REGISTROS
C07D DD7707             LD    (IX+7) ,A     ;insere A no 7º elemento
C080 060F               LD    B.15          ;contador de loop
C082 0E00               LD    C,0           ;contador de registros
C084             LOOPEN:                    ;
C084 59                 LD    E.C           ;carrega E com nº do registro
C085 CDF6C0             CALL  IMPHEX        ;mostra nº do registro
C088 3E20               LD    A, ' '        ;
C08A CDA200             CALL  CHPUT         ;espaço em branco
C08D DD7E00             LD    A,(IX)        ;carrega A com o valor do re-
C090 5F                 LD    E.A           ;gistro e o insere em E
C091 79                 LD    A.C           ;carrega A com o nº do canal
C092 DD23               INC   IX            ;incrementa IX
C094 0C                 INC   C             ;incrementa o contador
C095 CD9300             CALL  WRTPSG        ;escreve num registro do PSG
C098 CDF6C0             CALL  IMPHEX        ;imprime o valor do registro
C09B 3E0A               LD    A, 0AH        ;muda de linha
C09D CDA200             CALL  0A2H          ;
C0A0 3E0D               LD    A, 0DH        ;
C0A2 CDA200             CALL  0A2H          ;
C0A5 10DD               DJNZ  LOOPEN        ;
C0A7 C9                 RET                 ;efetua LOOP
C0A8             MUDCA:                     ;
C0A8 3E0A               LD    A.0AH         ;muda de linha
C0AA CDA200             CALL  CHPUT         ;
C0AD 3E0D               LD    A, 0DH        ;
C0AF CDA200             CALL  CHPUT         ;
C0B2 3A2AC1             LD    A,(REGOP)     ;carrega A com REGOP
C0B5 3C                 INC   A             ;incrementa A
C0B6 FE0E               CP    14            ;compara com 14
C0B8 D4C5C0             CALL  NC.ZERA       ;se for maior, salta para ZERA
C0BB 322AC1             LD    (REGOP),A     ;atualiza variável
C0BE 5F                 LD    E.A           ;imprime nº de canal
C0BF CDF6C0             CALL  IMPHEX        ;
C0C2 C327C0             JP    LOOPRI        ;salta para LOOPRI
C0C5             ZERA:                      ;
C0C5 3E00               LD    A.0           ;carrega 0 em A
C0C7 C9                 RET                 ;retorna
C0C8             ALTMIX:                    ;
C0C8 210909             LD    HL.0909H      ;escreve MIX
C0CB CDC600             CALL  POSIT         ;a partir da posição (9,9)
C0CE 3E4D               LD    A,'M'         ;
C0D0 CDA200             CALL  CHPUT         ;
C0D3 3E49               LD    A,'I'         ;
C0D5 CDA200             CALL  CHPUT         ;
C0D8 3E58               LD    A,'X'         ;
C0DA CDA200             CALL  CHPUT         ;
C0DD 0608               LD    B,08          ;
C0DF             LOOP:                      ;
C0DF CD9F00             CALL  CHGET         ;lê uma tecla
C0E2 CDA200             CALL  CHPUT         ;imprime tecla pressionada
C0E5 CB22               SLA   D             ;roda D
C0E7 FE30               CP    '0'           ;compara tecla com 0
C0E9 2802               JR    Z,CONT1       ;se igual, salta para CONT1
C0EB CBC2               SET   0,D           ;seta bit 0 de D
C0ED             CONT1:                     ;
```

```
C0ED 10F0               DJNZ LOOP           ;efetua LOOP
C0EF 7A                 LD   A,D            ;carrega A com novo valor
C0F0 3228C1             LD   (MISTUR),A     ;atualiza variável
C0F3 C327C0             JP   LOOPRI         ;salta para LOOPRI
C0F6                IMPHEX:                 ;
C0F6 7B                 LD   A,E            ;deixa  A com o
C0F7 CB3F               SRL  A              ;nibble mais
C0F9 CB3F               SRL  A              ;significativo de E
C0FB CB3F               SRL  A              ;
C0FD CB3F               SRL  A              ;
C0FF 21CCC1             LD   HL,TABHEX      ;aponta HL para TABHEX
C102 D5                 PUSH DE             ;guarda valor no STACK
C103 5F                 LD   E,A            ;aponta HL para o código
C104 1600               LD   D,0            ;ASCII correspondente
C106 19                 ADD  HL,DE          ;
C107 7E                 LD   A,(HL)         ;imprime o dígito
C108 CDA200             CALL CHPUT          ;mais significativo
C10B D1                 POP  DE             ;insere o topo do STACK em DE
C10C 21CCC1             LD   HL,TABHEX      ;aponta HL para TABHEX
C10F 7B                 LD   A,E            ;deixa em A apenas o nibble
C110 E60F               AND  00001111B      ;menos significativo de E
C112 5F                 LD   E,A            ;aponta HL para o código
C113 1600               LD   D,0            ;correspondente ao dígito
C115 19                 ADD  HL,DE          ;
C116 7E                 LD   A,(HL)         ;imprime o dígito menos
C117 CDA200             CALL CHPUT          ;significativo
C11A C9                 RET                 ;retorna
C11B          REGISTROS:DEFS 8
C123 0F0F0F00           DEFB 15,15,15,0,0,0,0
C127 000000
C12A 38           REGOP: DEFB 00111000B
C12B 00           MISTUR:DEFB 0
C12C 30313233 TABHEX:DEFB '0123456789ABCDEF'
C130 34353637
C134 38394142
C138 43444546
C13C 5245472E TABNOM:DEFB 'REG. VALOR'
C140 2056414C
C144 4F52
C146                    END
```

A figura 9.12 irá ajudá-lo a programar os registros do PSG. Em caso de dúvida consulte o livro "Aprofundando-se no MSX".

FIGURA 9.12 - Registros do PSG .

	VALOR MÁXIMO		BIT	7	6	5	4	3	2	1	0
0	255	PERÍODO DO CANAL A	L (8 bits)								
1	15		H (4 bits)								
2	255	PERÍODO DO CANAL B	L (8 bits)								
3	15		H (4 bits)								
4	255	PERÍODO DO CANAL C	L (8 bits)								
5	15		H (4 bits)								
6	31	PERÍODO DO RUÍDO	PR (5 bits)								
7	63	MISTURADOR	habilitam (0) cada canal	USADOS P/ JOYSTICK		RUÍDO C	RUÍDO B	RUÍDO A	TONS C	TONS B	TONS A
8	15	VOLUME DO CANAL A	Quando M está ativado (8-16) o volume é controlado pelo envelope				M				
9	15	VOLUME DO CANAL B					M				
10	15	VOLUME DO CANAL C					M				
11	255	PERÍODO DO ENVELOPE	L (8 bits)								
12	255		H (8 bits)								
13	14	FORMA DO ENVELOPE	veja tabela					E3	E2	E1	E0

Para finalizarmos, apresentaremos um prorama bastante simples e que utiliza a interrupção do micro.

Quando o processador de vídeo interrompe o Z-80 a cada 60 Hz, para serem efetuadas a varredura do teclado e outras tarefas, o endereço FD9FH é chamado com uma instrução CALL. Esse endereço, normalmente contém um C9H (RET) e portanto a execução volta para a ROM. Note, entretanto, que esse endereço se situa na RAM e portanto é variável.

Portanto nós alteraremos esse endereço colocando uma instrução do tipo:

```
JP XXXX
```

onde XXXX será o início de uma pequena sub-rotina.

A figura 9.13 apresenta um pequeno programa que incrementa a coordenada do SPRITE da camada 0.

FIGURA 9.13 - Programa de movimentação de um SPRITE.

```
C000 210CC0          LD    HL,INICIO   ;
C003 22A0FD          LD    (HOOK+1),HL ;altera vetor
C006 3EC3            LD    A,0C3H      ;    da
C008 329FFD          LD    (HOOK),A    ;interrupção
C00B C9              RET               ;
C00C          INICIO:                  ;
C00C F3              DI                ;desabilita a
C00D F5              PUSH  AF          ;interrupção
C00E E5              PUSH  HL          ;
C00F 21001B          LD    HL,1800H    ;
C012 CD4A00          CALL  RDVRM       ;lê byte da VRAM
C015 3C              INC   A           ;incrementa
C016 CD4D00          CALL  WRVRM       ;atualiza a VRAM
C019 E1              POP   HL          ;
C01A F1              POP   AF          ;
C01B FB              EI                ;habilita a
C01C C9              RET               ;interrupção
C01D                 END
```

Execute-o com o programa em BASIC da figura 9.14

FIGURA 9.14 - Programa auxiliar em BASIC.

```
10 SCREEN 1
20 SPRITE$(1)="AAAAAAAA"
30 PUTSPRITE 0,(128,96),13,1
40 DEFUSR=&HC000
50 PRINTUSR(0)
60 END
```

Você poderá encontrar muitas outras aplicações da Linguagem de Máquina nos livros "Programação Avançada em MSX" e "Aprofundando-se no MSX". Nesses livros, detalhes e sofisticações são explicados e exemplificados. Programas prontos para serem usados você pode encontrar no livro "TRON e TROFF dicas e macetes para o MSX". (Lançamento em 11/87).

EXERCíCIOS

1-) Faça um programa que, dada uma STRING, imprima-a na tela gráfica do final para o início.

2-) Modifique o programa do exercício 1 para que ele imprima na impressora ao mesmo tempo em que imprime no vídeo.

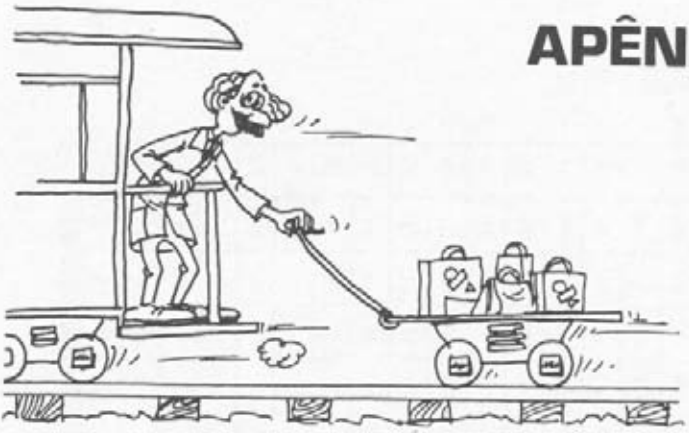
3-) Elabore um programa que, dado um número inteiro como parâmetro, copie toda a VRAM na página indicada.

4-) Junte ao programa do exercício 3 outro programa que, dado um número inteiro como parâmetro, copie os 16 K daquela página na VRAM.

DICA: Nos exercícios 3 e 4 utilize as rotinas LDIRVM e LDIRMV do BIOS.

5-) Faça um programa que envie a tabela de padrões da SCREEN 2 para a impressora (que deve operar em modo gráfico).
OBS.: Não esqueça de enviar os caracteres 0DH e 0AH para a impressora avançar de linha.

APÊNDICES



APÊNDICE 1 - Conversão de Sistemas de Numeração

A tabela localizada na próxima página permi-fazer a rápida conversão dos sistemas de numeração binário (base 2), decimal (base 10) e hexadecimal (base 16), para números que vão de 0 a 255 (máximo conteúdo de um byte de 8 bits). Para exemplificar seu uso, vamos imaginar um byte constituido dos seguintes algarismos binários: 11011010

Este byte deve ser dividido em duas metades (nibbles). A primeira metade é a mais significativa (MSN) e a segunda é a menos significativa (LSN):

1101(MSN) 1010 (LSN)

Na tabela de conversão as linhas correspondem aos MSN. No nosso exemplo a linha 1101 corresponde ao dígito D em hexadecimal. Analogamente, as colunas correspondem à segunda metade (LSN). Ainda no nosso exemplo, a coluna 1010 equivale ao dígito A em hexadecimal. No cruzamento da linha 1101 (D) com a coluna 1010 (A), temos o número 218. Portanto:

11011010 (binário) = DA (hexadecimal) = 218 (decimal)

Obviamente o próprio computador pode fazer esta conversão bastando lembrar que os números em binário devem ser precedidos por &B e em hexadecimal por &H. Experimente digitar e executar estas instruções:

```
PRINT &HDA
PRINT &B11011010
PRINT RIGHT$("0000000"+BIN$(218),8)
PRINT RIGHT$("0000000"+BIN$(&HDA),8)
PRINT RIGHT$("0"+HEX$(218),2)
PRINT RIGHT$("0"+HEX$(&B11011010),2)
```

2º METADE *LSN*

1ª METADE *MSN*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0000
	1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0001
	2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	0010
	3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	0011
	4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	0100
	5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	0101
	6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	0110
	7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	0111
	8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	1000
	9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	1001
	A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	1010
	B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	1011
	C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	1100
	D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	1101
	E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	1110
	F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	1111
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	

APÊNDICE 2 - TABELA DAS INSTRUÇÕES DO Z80

Este apêndice apresenta uma tabela de todas as instruções da linguagem de máquina disponíveis para o microprocessador do MSX (Z80).

Para cada instrução é apresentado o mnemônico, o número de bytes, as flags afetadas e o seu código hexadecimal.

Quanto às flags, são apresentadas na seguinte ordem:

C Z S P/O AC N

A notação utilizada é :

- : flag não afetada; permanece em seu valor anterior à instrução.
- ? : flag afetada; resultado indefinido.
- * : flag afetada; resultado depende da operação.
- 0 : flag afetada; resultado zero.
- 1 : flag afetada; resultado um.
- o : flag P/O afetada para indicar presença ou não de OVERFLOW.
- p : flag P/O afetada para indicar PARIDADE.
- t : o conteúdo do FLIP-FLOP de interrupção é copiado na flag P/O.

Além disso utilizam-se as seguintes notações genéricas:

- nn : número hexadecimal de um byte.
- nnmm : número hexadecimal de dois bytes.
- ss : número hexadecimal de um byte com sinal (-127 a 128).

MNEMÔNICO	B	FLAGS	CÓDIGO
LD A,A	1	------	7F
LD A,B	1	------	78
LD A,C	1	------	79
LD A,D	1	------	7A
LD A,E	1	------	7B
LD A,H	1	------	7C
LD A,L	1	------	7D

MNEMÔNICO	B	FLAGS	CÓDIGO
LD B,A	1	------	47
LD B,B	1	------	40
LD B,C	1	------	41
LD B,D	1	------	42
LD B,E	1	------	43
LD B,H	1	------	44
LD B,L	1	------	45

MNEMÔNICO	B	FLAGS	CÓDIGO
LD C,A	1	------	4F
LD C,B	1	------	48
LD C,C	1	------	49
LD C,D	1	------	4A
LD C,E	1	------	4B
LD C,H	1	------	4C
LD C,L	1	------	4D
LD D,A	1	------	57
LD D,B	1	------	50
LD D,C	1	------	51
LD D,D	1	------	52
LD D,E	1	------	53
LD D,H	1	------	54
LD D,L	1	------	55
LD E,A	1	------	5F
LD E,B	1	------	58
LD E,C	1	------	59
LD E,D	1	------	5A
LD E,E	1	------	5B
LD E,H	1	------	5C
LD E,L	1	------	5D
LD H,A	1	------	67
LD H,B	1	------	60
LD H,C	1	------	61
LD H,D	1	------	62
LD H,E	1	------	63
LD H,H	1	------	64
LD H,L	1	------	65
LD L,A	1	------	6F
LD L,B	1	------	68
LD L,C	1	------	69
LD L,D	1	------	6A
LD L,E	1	------	6B
LD L,H	1	------	6C
LD L,L	1	------	6D
LD A,I	2	-**t00	ED57
LD I,A	2	------	ED47
LD A,R	2	-**t00	ED5F
LD R,A	2	------	ED4F
LD SP,HL	1	------	F9
LD SP,IX	2	------	DDF9
LD SP,IY	2	------	FDF9
LD A,nn	2	------	3Enn
LD B,nn	2	------	06nn

MNEMÔNICO	B	FLAGS	CÓDIGO
LD C,nn	2	------	0Enn
LD D,nn	2	------	16nn
LD E,nn	2	------	1Enn
LD H,nn	2	------	26nn
LD L,nn	2	------	2Enn
LD BC,nnmm	3	------	01mmnn
LD DE,nnmm	3	------	11mmnn
LD HL,nnmm	3	------	21mmnn
LD IX,nnmm	4	------	DD21mmnn
LD IY,nnmm	4	------	FD21mmnn
LD SP,nnmm	3	------	31mmnn
LD (HL),nn	2	------	36nn
LD (IX+ss),nn	4	------	DD36ssnn
LD (IY+ss),nn	4	------	FD36ssnn
LD A,(BC)	1	------	0A
LD A,(DE)	1	------	1A
LD A,(HL)	1	------	7E
LD A,(IX+ss)	3	------	DD7Ess
LD A,(IY+ss)	3	------	FD7Ess
LD B,(HL)	1	------	46
LD B,(IX+ss)	3	------	DD46ss
LD B,(IY+ss)	3	------	FD46ss
LD C,(HL)	1	------	4E
LD C,(IX+ss)	3	------	DD4Ess
LD C,(IY+ss)	3	------	FD4Ess
LD D,(HL)	1	------	56
LD D,(IX+ss)	3	------	DD56ss
LD D,(IY+ss)	3	------	FD56ss
LD E,(HL)	1	------	5E
LD E,(IX+ss)	3	------	DD5Ess
LD E,(IY+ss)	3	------	FD5Ess
LD H,(HL)	1	------	66
LD H,(IX+ss)	3	------	DD66ss
LD H,(IY+ss)	3	------	FD66ss
LD L,(HL)	1	------	6E
LD L,(IX+ss)	3	------	DD6Ess
LD L,(IY+ss)	3	------	FD6Ess
LD (BC),A	1	------	02
LD (DE),A	1	------	12
LD (HL),A	1	------	77
LD (IX+ss),A	3	------	DD77ss
LD (IY+ss),A	3	------	FD77ss
LD (HL),B	1	------	70
LD (IX+ss),B	3	------	DD70ss

MNEMÔNICO	B	FLAGS	CÓDIGO
LD (IY+ss),B	3	------	FD70ss
LD (HL),C	1	------	71
LD (IX+ss),C	3	------	DD71ss
LD (IY+ss),C	3	------	FD71ss
LD (HL),D	1	------	72
LD (IX+ss),D	3	------	DD72ss
LD (IY+ss),D	3	------	FD72ss
LD (HL),E	1	------	73
LD (IX+ss),E	3	------	DD73ss
LD (IY+ss),E	3	------	FD73ss
LD (HL),H	1	------	74
LD (IX+ss),H	3	------	DD74ss
LD (IY+ss),H	3	------	FD74ss
LD (HL),L	1	------	75
LD (IX+ss),L	3	------	DD75ss
LD (IY+ss),L	3	------	FD75ss
LD A,(nnmm)	3	------	3Ammnn
LD BC,(nnmm)	4	------	ED4Bmmnn
LD DE,(nnmm)	4	------	ED5Bmmnn
LD HL,(nnmm)	3	------	2Ammnn
LD IX,(nnmm)	4	------	DD2Ammnn
LD IY,(nnmm)	4	------	FD2Ammnn
LD (nnmm),A	3	------	32mmnn
LD (nnmm),BC	4	------	ED43mmnn
LD (nnmm),DE	4	------	ED53mmnn
LD (nnmm),HL	3	------	22mmnn
LD (nnmm),IX	4	------	DD22mmnn
LD (nnmm),IY	4	------	FD22mmnn
LD SP,(nnmm)	4	------	ED7Bmmnn
LD (nnmm),SP	4	------	ED73mmnn
LDD	2	---*00	EDA8
LDDR	2	---000	EDB8
LDI	2	---*00	EDA0
LDIR	2	---000	EDB0
ADD A,A	1	***o*0	87
ADD A,B	1	***o*0	80
ADD A,C	1	***o*0	81
ADD A,D	1	***o*0	82
ADD A,E	1	***o*0	83
ADD A,H	1	***o*0	84
ADD A,L	1	***o*0	85
ADD A,(HL)	1	***o*0	86
ADD A,(IX+ss)	3	***o*0	DD86ss
ADD A,(IY+ss)	3	***o*0	FD86ss

MNEMÔNICO	B	FLAGS	CÓDIGO
ADC A,A	1	***o*0	8F
ADC A,B	1	***o*0	88
ADC A,C	1	***o*0	89
ADC A,D	1	***o*0	8A
ADC A,E	1	***o*0	8B
ADC A,H	1	***o*0	8C
ADC A,L	1	***o*0	8D
ADC A,(HL)	1	***o*0	8E
ADC A,(IX+ss)	3	***o*0	DD8Ess
ADC A,(IY+ss)	3	***o*0	FD8Ess
ADD HL,BC	1	*---?0	09
ADD HL,DE	1	*---?0	19
ADD HL,HL	1	*---?0	29
ADD IX,BC	2	*---?0	DD09
ADD IX,DE	2	*---?0	DD19
ADD IX,IX	2	*---?0	DD29
ADD IY,BC	2	*---?0	FD09
ADD IY,DE	2	*---?0	FD19
ADD IY,IY	2	*---?0	FD29
ADD HL,SP	1	*---?0	39
ADD IX,SP	2	*---?0	DD39
ADD IY,SP	2	*---?0	FD39
ADC HL,BC	2	***o?0	ED4A
ADC HL,DE	2	***o?0	ED5A
ADC HL,HL	2	***o?0	ED6A
ADC HL,SP	2	***o?0	ED7A
ADD A,nn	2	***o*0	C6nn
ADC A,nn	2	***o*0	CEnn
SUB A,A	1	***o*1	97
SUB A,B	1	***o*1	90
SUB A,C	1	***o*1	91
SUB A,D	1	***o*1	92
SUB A,E	1	***o*1	93
SUB A,H	1	***o*1	94
SUB A,L	1	***o*1	95
SUB A,(HL)	1	***o*1	96
SUB A,(IX+ss)	3	***o*1	DD96ss
SUB A,(IY+ss)	3	***o*1	FD96ss
SBC A,A	1	***o*1	9F
SBC A,B	1	***o*1	98
SBC A,C	1	***o*1	99
SBC A,D	1	***o*1	9A
SBC A,E	1	***o*1	9B
SBC A,H	1	***o*1	9C

MNEMÔNICO	B	FLAGS	CÓDIGO
SBC A,L	1	***o*i	9D
SBC A,(HL)	1	***o*i	9E
SBC A,(IX+ss)	3	***o*i	DD9Ess
SBC A,(IY+ss)	3	***o*i	FD9Ess
SBC HL,BC	2	***o?i	ED42
SBC HL,DE	2	***o?i	ED52
SBC HL,HL	2	***o?i	ED62
SBC HL,SP	2	***o?i	ED72
SUB A,nn	2	***o*i	D6nn
SBC A,nn	2	***o*i	DEnn
INC A	1	-**o*0	3C
INC B	1	-**o*0	04
INC C	1	-**o*0	0C
INC D	1	-**o*0	14
INC E	1	-**o*0	1C
INC H	1	-**o*0	24
INC L	1	-**o*0	2C
INC BC	1	------	03
INC DE	1	------	13
INC HL	1	------	23
INC IX	2	------	DD23
INC IY	2	------	FD23
INC SP	1	------	33
INC (HL)	1	-**o*0	34
INC (IX+ss)	3	-**o*0	DD34ss
INC (IY+ss)	3	-**o*0	FD34ss
DEC A	1	-**o*i	3D
DEC B	1	-**o*i	05
DEC C	1	-**o*i	0D
DEC D	1	-**o*i	15
DEC E	1	-**o*i	1D
DEC H	1	-**o*i	25
DEC L	1	-**o*i	2D
DEC BC	1	------	0B
DEC DE	1	------	1B
DEC HL	1	------	2B
DEC IX	2	------	DD2B
DEC IY	2	------	FD2B
DEC SP	1	------	3B
DEC (HL)	1	-**o*i	35
DEC (IX+ss)	3	-**o*i	DD35ss
DEC (IY+ss)	3	-**o*i	FD35ss
NEG	2	***o*i	ED44
PUSH AF	1	------	F5

MNEMÔNICO	B	FLAGS	CÓDIGO
PUSH BC	1	------	C5
PUSH DE	1	------	D5
PUSH HL	1	------	E5
PUSH IX	2	------	DDE5
PUSH IY	2	------	FDE5
POP AF	1	------	F1
POP BC	1	------	C1
POP DE	1	------	D1
POP HL	1	------	E1
POP IX	2	------	DDE1
POP IY	2	------	FDE1
LD SP,HL	1	------	F9
LD SP,IX	2	------	DDF9
LD SP,IY	2	------	FDF9
LD SP,nnmm	3	------	31mmnn
LD SP,(nnmm)	4	------	ED7Bmmnn
LD (nnmm),SP	4	------	ED73mmnn
ADD HL,SP	1	*---?0	39
ADD IX,SP	2	*---?0	DD39
ADD IY,SP	2	*---?0	FD39
ADC HL,SP	2	***o?0	ED7A
SBC HL,SP	2	***o?i	ED72
INC SP	1	------	33
DEC SP	1	------	3B
EX (SP),HL	1	------	E3
EX (SP),IX	2	------	DDE3
EX (SP),IY	2	------	FDE3
JP nnmm	3	------	C3mmnn
JP (HL)	1	------	E9
JP (IX)	2	------	DDE9
JP (IY)	2	------	FDE9
JP C,nnmm	3	------	DAmmnn
JP NC,nnmm	3	------	D2mmnn
JP Z,nnmm	3	------	CAmmnn
JP NZ,nnmm	3	------	C2mmnn
JP PE,nnmm	3	------	EAmmnn
JP PO,nnmm	3	------	E2mmnn
JP M,nnmm	3	------	FAmmnn
JP P,nnmm	3	------	F2mmnn
JR ss	2	------	18ss
JR C,ss	2	------	38ss
JR NC,ss	2	------	30ss
JR Z,ss	2	------	28ss
JR NZ,ss	2	------	20ss

MNEMÔNICO	B	FLAGS	CÓDIGO
DJNZ ss	2	------	10ss
CALL nnmm	3	------	CDmmnn
CALL C,nnmm	3	------	DCmmnn
CALL NC,nnmm	3	------	D4mmnn
CALL Z,nnmm	3	------	CCmmnn
CALL NZ,nnmm	3	------	C4mmnn
CALL PE,nnmm	3	------	ECmmnn
CALL PO,nnmm	3	------	E4mmnn
CALL M,nnmm	3	------	FCmmnn
CALL P,nnmm	3	------	F4mmnn
RET	1	------	C9
RET C	1	------	D8
RET NC	1	------	D0
RET Z	1	------	C8
RET NZ	1	------	C0
RET PE	1	------	E8
RET PO	1	------	E0
RET M	1	------	F8
RET P	1	------	F0
RETI	2	------	ED4D
RETN	2	------	ED45
RST 0	1	------	C7
RST 8	1	------	CF
RST 16	1	------	D7
RST 24	1	------	DF
RST 32	1	------	E7
RST 40	1	------	EF
RST 48	1	------	F7
RST 56	1	------	FF
CP A	1	***o*1	BF
CP B	1	***o*1	B8
CP C	1	***o*1	B9
CP D	1	***o*1	BA
CP E	1	***o*1	BB
CP H	1	***o*1	BC
CP L	1	***o*1	BD
CP (HL)	1	***o*1	BE
CP (IX+ss)	3	***o*1	DDBEss
CP (IY+ss)	3	***o*1	FDBEss
CP nn	2	***o*1	FEnn
CPD	2	-****1	EDA9
CPDR	2	-****1	EDB9
CPI	2	-****1	EDA1
CPIR	2	-****1	EDB1

MNEMÔNICO	B	FLAGS	CÓDIGO
AND A	1	0**p10	A7
AND B	1	0**p10	A0
AND C	1	0**p10	A1
AND D	1	0**p10	A2
AND E	1	0**p10	A3
AND H	1	0**p10	A4
AND L	1	0**p10	A5
AND (HL)	1	0**p10	A6
AND (IX+ss)	3	0**p10	DDA6ss
AND (IY+ss)	3	0**p10	FDA6ss
AND nn	2	0**p10	E6nn
OR A	1	0**p10	B7
OR B	1	0**p10	B0
OR C	1	0**p10	B1
OR D	1	0**p10	B2
OR E	1	0**p10	B3
OR H	1	0**p10	B4
OR L	1	0**p10	B5
OR (HL)	1	0**p10	B6
OR (IX+ss)	3	0**p10	DDB6ss
OR (IY+ss)	3	0**p10	FDB6ss
OR nn	2	0**p10	F6nn
XOR A	1	0**p10	AF
XOR B	1	0**p10	A8
XOR C	1	0**p10	A9
XOR D	1	0**p10	AA
XOR E	1	0**p10	AB
XOR H	1	0**p10	AC
XOR L	1	0**p10	AD
XOR (HL)	1	0**p10	AE
XOR (IX+ss)	3	0**p10	DDAEss
XOR (IY+ss)	3	0**p10	FDAEss
XOR nn	2	0**p10	EEnn
BIT 0,A	2	-*??10	CB47
BIT 0,B	2	-*??10	CB40
BIT 0,C	2	-*??10	CB41
BIT 0,D	2	-*??10	CB42
BIT 0,E	2	-*??10	CB43
BIT 0,H	2	-*??10	CB44
BIT 0,L	2	-*??10	CB45
BIT 1,A	2	-*??10	CB4F
BIT 1,B	2	-*??10	CB48
BIT 1,C	2	-*??10	CB49
BIT 1,D	2	-*??10	CB4A

MNEMÔNICO	B	FLAGS	CÓDIGO
BIT 1,E	2	-*??10	CB4B
BIT 1,H	2	-*??10	CB4C
BIT 1,L	2	-*??10	CB4D
BIT 2,A	2	-*??10	CB57
BIT 2,B	2	-*??10	CB50
BIT 2,C	2	-*??10	CB51
BIT 2,D	2	-*??10	CB52
BIT 2,E	2	-*??10	CB53
BIT 2,H	2	-*??10	CB54
BIT 2,L	2	-*??10	CB55
BIT 3,A	2	-*??10	CB5F
BIT 3,B	2	-*??10	CB58
BIT 3,C	2	-*??10	CB59
BIT 3,D	2	-*??10	CB5A
BIT 3,E	2	-*??10	CB5B
BIT 3,H	2	-*??10	CB5C
BIT 3,L	2	-*??10	CB5D
BIT 4,A	2	-*??10	CB67
BIT 4,B	2	-*??10	CB60
BIT 4,C	2	-*??10	CB61
BIT 4,D	2	-*??10	CB62
BIT 4,E	2	-*??10	CB63
BIT 4,H	2	-*??10	CB64
BIT 4,L	2	-*??10	CB65
BIT 5,A	2	-*??10	CB6F
BIT 5,B	2	-*??10	CB68
BIT 5,C	2	-*??10	CB69
BIT 5,D	2	-*??10	CB6A
BIT 5,E	2	-*??10	CB6B
BIT 5,H	2	-*??10	CB6C
BIT 5,L	2	-*??10	CB6D
BIT 6,A	2	-*??10	CB77
BIT 6,B	2	-*??10	CB70
BIT 6,C	2	-*??10	CB71
BIT 6,D	2	-*??10	CB72
BIT 6,E	2	-*??10	CB73
BIT 6,H	2	-*??10	CB74
BIT 6,L	2	-*??10	CB75
BIT 7,A	2	-*??10	CB7F
BIT 7,B	2	-*??10	CB78
BIT 7,C	2	-*??10	CB79
BIT 7,D	2	-*??10	CB7A
BIT 7,E	2	-*??10	CB7B
BIT 7,H	2	-*??10	CB7C

MNEMÔNICO	B	FLAGS	CÓDIGO
BIT 7,L	2	-*??10	CB7C
BIT 0,(HL)	2	-*??10	CB46
BIT 0,(IX+ss)	4	-*??10	DDCBss46
BIT 0,(IY+ss)	4	-*??10	FDCBss46
BIT 1,(HL)	2	-*??10	CB4E
BIT 1,(IX+ss)	4	-*??10	DDCBss4E
BIT 1,(IY+ss)	4	-*??10	FDCBss4E
BIT 2,(HL)	2	-*??10	CB56
BIT 2,(IX+ss)	4	-*??10	DDCBss56
BIT 2,(IY+ss)	4	-*??10	FDCBss56
BIT 3,(HL)	2	-*??10	CB5E
BIT 3,(IX+ss)	4	-*??10	DDCBss5E
BIT 3,(IY+ss)	4	-*??10	FDCBss5E
BIT 4,(HL)	2	-*??10	CB66
BIT 4,(IX+ss)	4	-*??10	DDCBss66
BIT 4,(IY+ss)	4	-*??10	FDCBss66
BIT 5,(HL)	2	-*??10	CB6E
BIT 5,(IX+ss)	4	-*??10	DDCBss6E
BIT 5,(IY+ss)	4	-*??10	FDCBss6E
BIT 6,(HL)	2	-*??10	CB76
BIT 6,(IX+ss)	4	-*??10	DDCBss76
BIT 6,(IY+ss)	4	-*??10	FDCBss76
BIT 7,(HL)	2	-*??10	CB7E
BIT 7,(IX+ss)	4	-*??10	DDCBss7E
BIT 7,(IY+ss)	4	-*??10	FDCBss7E
SET 0,A	2	------	CBC7
SET 0,B	2	------	CBC0
SET 0,C	2	------	CBC1
SET 0,D	2	------	CBC2
SET 0,E	2	------	CBC3
SET 0,H	2	------	CBC4
SET 0,L	2	------	CBC5
SET 1,A	2	------	CBCF
SET 1,B	2	------	CBC8
SET 1,C	2	------	CBC9
SET 1,D	2	------	CBCA
SET 1,E	2	------	CBCB
SET 1,H	2	------	CBCC
SET 1,L	2	------	CBCD
SET 2,A	2	------	CBD7
SET 2,B	2	------	CBD0
SET 2,C	2	------	CBD1
SET 2,D	2	------	CBD2
SET 2,E	2	------	CBD3

MNEMÔNICO	B	FLAGS	CÓDIGO
SET 2,H	2	------	CBD4
SET 2,L	2	------	CBD5
SET 3,A	2	------	CBDF
SET 3,B	2	------	CBD8
SET 3,C	2	------	CBD9
SET 3,D	2	------	CBDA
SET 3,E	2	------	CBDB
SET 3,H	2	------	CBDC
SET 3,L	2	------	CBDD
SET 4,A	2	------	CBE7
SET 4,B	2	------	CBE0
SET 4,C	2	------	CBE1
SET 4,D	2	------	CBE2
SET 4,E	2	------	CBE3
SET 4,H	2	------	CBE4
SET 4,L	2	------	CBE5
SET 5,A	2	------	CBEF
SET 5,B	2	------	CBE8
SET 5,C	2	------	CBE9
SET 5,D	2	------	CBEA
SET 5,E	2	------	CBEB
SET 5,H	2	------	CBEC
SET 5,L	2	------	CBED
SET 6,A	2	------	CBF7
SET 6,B	2	------	CBF0
SET 6,C	2	------	CBF1
SET 6,D	2	------	CBF2
SET 6,E	2	------	CBF3
SET 6,H	2	------	CBF4
SET 6,L	2	------	CBF5
SET 7,A	2	------	CBFF
SET 7,B	2	------	CBF8
SET 7,C	2	------	CBF9
SET 7,D	2	------	CBFA
SET 7,E	2	------	CBFB
SET 7,H	2	------	CBFC
SET 7,L	2	------	CBFD
SET 0,(HL)	2	------	CBC6
SET 0,(IX+ss)	4	------	DDCBssC6
SET 0,(IY+ss)	4	------	FDCBssC6
SET 1,(HL)	2	------	CBCE
SET 1,(IX+ss)	4	------	DDCBssCE
SET 1,(IY+ss)	4	------	FDCBssCE
SET 2,(HL)	2	------	CBD6

MNEMÔNICO	B	FLAGS	CÓDIGO
SET 2,(IX+ss)	4	------	DDCBssD6
SET 2,(IY+ss)	4	------	FDCBssD6
SET 3,(HL)	2	------	CBDE
SET 3,(IX+ss)	4	------	DDCBssDE
SET 3,(IY+ss)	4	------	FDCBssDE
SET 4,(HL)	2	------	CBE6
SET 4,(IX+ss)	4	------	DDCBssE6
SET 4,(IY+ss)	4	------	FDCBssE6
SET 5,(HL)	2	------	CBEE
SET 5,(IX+ss)	4	------	DDCBssEE
SET 5,(IY+ss)	4	------	FDCBssEE
SET 6,(HL)	2	------	CBF6
SET 6,(IX+ss)	4	------	DDCBssF6
SET 6,(IY+ss)	4	------	FDCBssF6
SET 7,(HL)	2	------	CBFE
SET 7,(IX+ss)	4	------	DDCBssFE
SET 7,(IY+ss)	4	------	FDCBssFE
RES 0,A	2	------	CB87
RES 0,B	2	------	CB80
RES 0,C	2	------	CB81
RES 0,D	2	------	CB82
RES 0,E	2	------	CB83
RES 0,H	2	------	CB84
RES 0,L	2	------	CB85
RES 1,A	2	------	CB8F
RES 1,B	2	------	CB88
RES 1,C	2	------	CB89
RES 1,D	2	------	CB8A
RES 1,E	2	------	CB8B
RES 1,H	2	------	CB8C
RES 1,L	2	------	CB8D
RES 2,A	2	------	CB97
RES 2,B	2	------	CB90
RES 2,C	2	------	CB91
RES 2,D	2	------	CB92
RES 2,E	2	------	CB93
RES 2,H	2	------	CB94
RES 2,L	2	------	CB95
RES 3,A	2	------	CB9F
RES 3,B	2	------	CB98
RES 3,C	2	------	CB99
RES 3,D	2	------	CB9A
RES 3,E	2	------	CB9B
RES 3,H	2	------	CB9C

MNEMÔNICO	B	FLAGS	CÓDIGO
RES 3,L	2	------	CB9D
RES 4,A	2	------	CBA7
RES 4,B	2	------	CBA0
RES 4,C	2	------	CBA1
RES 4,D	2	------	CBA2
RES 4,E	2	------	CBA3
RES 4,H	2	------	CBA4
RES 4,L	2	------	CBA5
RES 5,A	2	------	CBAF
RES 5,B	2	------	CBA8
RES 5,C	2	------	CBA9
RES 5,D	2	------	CBAA
RES 5,E	2	------	CBAB
RES 5,H	2	------	CBAC
RES 5,L	2	------	CBAD
RES 6,A	2	------	CBB7
RES 6,B	2	------	CBB0
RES 6,C	2	------	CBB1
RES 6,D	2	------	CBB2
RES 6,E	2	------	CBB3
RES 6,H	2	------	CBB4
RES 6,L	2	------	CBB5
RES 7,A	2	------	CBBF
RES 7,B	2	------	CBB8
RES 7,C	2	------	CBB9
RES 7,D	2	------	CBBA
RES 7,E	2	------	CBBB
RES 7,H	2	------	CBBC
RES 7,L	2	------	CBBD
RES 0,(HL)	2	------	CB86
RES 0,(IX+ss)	4	------	DDCBss86
RES 0,(IY+ss)	4	------	FDCBss86
RES 1,(HL)	2	------	CB8E
RES 1,(IX+ss)	4	------	DDCBss8E
RES 1,(IY+ss)	4	------	FDCBss8E
RES 2,(HL)	2	------	CB96
RES 2,(IX+ss)	4	------	DDCBss96
RES 2,(IY+ss)	4	------	FDCBss96
RES 3,(HL)	2	------	CB9E
RES 3,(IX+ss)	4	------	DDCBss9E
RES 3,(IY+ss)	4	------	FDCBss9E
RES 4,(HL)	2	------	CBA6
RES 4,(IX+ss)	4	------	DDCBssA6
RES 4,(IY+ss)	4	------	FDCBssA6

MNEMÔNICO	B	FLAGS	CÓDIGO
RES 5,(HL)	2	------	CBAE
RES 5,(IX+ss)	4	------	DDCBssAE
RES 5,(IY+ss)	4	------	FDCBssAE
RES 6,(HL)	2	------	CBB6
RES 6,(IX+ss)	4	------	DDCBssB6
RES 6,(IY+ss)	4	------	FDCBssB6
RES 7,(HL)	2	------	CBBE
RES 7,(IX+ss)	4	------	DDCBssBE
RES 7,(IY+ss)	4	------	FDCBssBE
RLC A	2	***p00	CB07
RLC B	2	***p00	CB00
RLC C	2	***p00	CB01
RLC D	2	***p00	CB02
RLC E	2	***p00	CB03
RLC H	2	***p00	CB04
RLC L	2	***p00	CB05
RLC (HL)	2	***p00	CB06
RLC (IX+ss)	4	***p00	DDCBss06
RLC (IY+ss)	4	***p00	FDCBss06
RRC A	2	***p00	CB0F
RRC B	2	***p00	CB08
RRC C	2	***p00	CB09
RRC D	2	***p00	CB0A
RRC E	2	***p00	CB0B
RRC H	2	***p00	CB0C
RRC L	2	***p00	CB0D
RRC (HL)	2	***p00	CB0E
RRC (IX+ss)	4	***p00	DDCBss0E
RRC (IY+ss)	4	***p00	FDCBss0E
RL A	2	***p00	CB17
RL B	2	***p00	CB10
RL C	2	***p00	CB11
RL D	2	***p00	CB12
RL E	2	***p00	CB13
RL H	2	***p00	CB14
RL L	2	***p00	CB15
RL (HL)	2	***p00	CB16
RL (IX+ss)	4	***p00	DDCBss16
RL (IY+ss)	4	***p00	FDCBss16
RR A	2	***p00	CB1F
RR B	2	***p00	CB18
RR C	2	***p00	CB19
RR D	2	***p00	CB1A
RR E	2	***p00	CB1B

MNEMÔNICO	B	FLAGS	CÓDIGO
RR H	2	***p00	CB1C
RR L	2	***p00	CB1D
RR (HL)	2	***p00	CB1E
RR (IX+ss)	4	***p00	DDCBssiE
RR (IY+ss)	4	***p00	FDCBssiE
SLA A	2	***p00	CB27
SLA B	2	***p00	CB20
SLA C	2	***p00	CB21
SLA D	2	***p00	CB22
SLA E	2	***p00	CB23
SLA H	2	***p00	CB24
SLA L	2	***p00	CB25
SLA (HL)	2	***p00	CB26
SLA (IX+ss)	4	***p00	DDCBss26
SLA (IY+ss)	4	***p00	FDCBss26
SRA A	2	***p00	CB2F
SRA B	2	***p00	CB28
SRA C	2	***p00	CB29
SRA D	2	***p00	CB2A
SRA E	2	***p00	CB2B
SRA H	2	***p00	CB2C
SRA L	2	***p00	CB2D
SRA (HL)	2	***p00	CB2E
SRA (IX+ss)	4	***p00	DDCBss2E
SRA (IY+ss)	4	***p00	FDCBss2E
SRL A	2	***p00	CB3F
SRL B	2	***p00	CB38
SRL C	2	***p00	CB39
SRL D	2	***p00	CB3A
SRL E	2	***p00	CB3B
SRL H	2	***p00	CB3C
SRL L	2	***p00	CB3D
SRL (HL)	2	***p00	CB3E
SRL (IX+ss)	4	***p00	DDCBss3E
SRL (IY+ss)	4	***p00	FDCBss3E
RLCA	1	*---00	07
RRCA	1	*---00	0F
RLA	1	*---00	17
RRA	1	*---00	1F
RLD	2	-**p00	ED6F
RRD	2	-**p00	ED67
CPL	1	----11	2F
CCF	1	*---?0	3F
SCF	1	1---00	37

MNEMÔNICO	B	FLAGS	CÓDIGO
EI	1	------	FB
DI	1	------	F3
IM 0	2	------	ED46
IM 2	2	------	ED56
IN A,nn	2	------	DBnn
IN A,(C)	2	-**p*0	ED78
IN B,(C)	2	-**p*0	ED40
IN C,(C)	2	-**p*0	ED48
IN D,(C)	2	-**p*0	ED50
IN E,(C)	2	-**p*0	ED58
IN H,(C)	2	-**p*0	ED60
IN L,(C)	2	-**p*0	ED68
IND	2	-*???1	EDAA
INDR	2	-1???1	EDBA
INI	2	-*???1	EDA2
INIR	2	-1???1	EDB2
OUT nn,A	2	------	D3nn
OUT (C),A	2	------	ED79
OUT (C),B	2	------	ED41
OUT (C),C	2	------	ED49
OUT (C),D	2	------	ED51
OUT (C),E	2	------	ED59
OUT (C),H	2	------	ED61
OUT (C),L	2	------	ED69
OUTD	2	-*???1	EDAB
OTDR	2	-1???1	EDBB
OUTI	2	-*???1	EDA3
OTIR	2	-1???1	EDB3
LD A,I	2	-**t00	ED57
LD I,A	2	------	ED47
EX AF,A',F'	1	------	08
EXX	1	------	D9
EX DE,HL	1	------	EB
EX (SP),HL	1	------	E3
EX (SP),IX	2	------	DDE3
EX (SP),IY	2	------	FDE3
NOP	1	------	00
HALT	1	------	76
DAA	1	***p*-	27
RDL	2	-**p00	ED6F
RRD	2	-**p00	ED67
LD A,R	2	-**t00	ED5F
LD R,A	2	------	ED4F

Bibliografia Aconselhada

APROFUNDANDO-SE NO MSX
Piazzi, Maldonado, Oliveira et al.
Para quem quer conhecer todos os detalhes da arquitetura do MSX: como usar os 32 Kb de RAM escondidos pela ROM, como redefinir caracteres, como usar o SOUND, como tirar cópias de telas gráficas na impressora, como fazer cópias de fitas. Todos os detalhes do MSX, o BIOS e as variáveis do sistema comentadas e um poderoso disassembler.

PROGRAMAÇÃO AVANÇADA EM MSX
Figueredo, Maldonado e Rossetto
Você saberia como compatibilizar sua impressora com o padrão de caracteres do MSX? Se sentiria à vontade para mudar os formatos de gravação em cassete para proteger seus programas e dados? Entender e projetar seus próprios programas em EPROM, instalados em cartuchos? Ou ainda usar rotinas e funções internas do seu MSX para aumentar a velocidade de seus programas ou obter efeitos especiais no monitor?
Se alguma destas perguntas tiver resposta negativa, você precisa ler este livro, dividido em duas partes: na primeira são abordados os detalhes íntimos do interpretador BASIC, desde a forma utilizada para a representação interna das linhas de comandos e dados na memória, até a identificação dos pontos de acesso às funções, rotinas e comandos do próprio MSX.
Na segunda parte são vistas todas as particularidades relativas ao tratamento das imagens no monitor de TV, são analisadas as técnicas de acesso, uso e controle das impressoras, cassetes e cartuchos (EPROM) e, por fim, como construir jogos em Assembly.

TABELA DE MNEMÔNICOS Z-80
Uma tabela com todos os mnemônicos do microprocessador Z-80 relacionados com seus códigos hexadecimais. Indispensável para quem está começando a programar em Linguagem de Máquina e não dispõe de um Assembler.

Para receber gratuitamente o boletim informativo da ALEPH, contendo dicas de programação, artigos técnicos e informações sobre os últimos lançamentos para seu micro, envie seu nome e endereço completos (incluindo o CEP) para a EDITORA ALEPH Caixa Postal 20.707 CEP 01498 São Paulo SP.

---

série didática **MSX**

A "Série Didática" é mais uma iniciativa pioneira da EDITORA ALEPH visando levar aos seus leitores conceitos específicos através de textos claros e, realmente, didáticos. O uso do microcomputador na educação pode ser desastroso ou extremamente proveitoso, dependendo da forma como ele for usado. Na "Série Didática" a utilização dos micros é o tema central, sendo repleta de exemplos de usos proveitosos.

linguagem de máquina

(MSX)

Conhecer a Linguagem de Máquina de um microprocessador desvenda-nos um horizonte inimaginavelmente vasto de usos e aplicações. Se o microprocessador for o Z-80, então o horizonte é ainda mais amplo. Basta verificar a diversidade de máquinas que se utilizam desse chip: os MSX, os TRS-80, os SINCLAIR, e até alguns mini-computadores.

Os autores deste livro são dois apaixonados usuários de microcomputadores que não se renderam às limitações do BASIC. FLAVIO ROSSINI, autor conhecido de todos os "SINCLAIRISTAS" brasileiros, e HENRIQUE DE FIGUEREDO LUZ, que dispensa apresentações aos usuários dos MSX são, além de tudo, excelentes professores. Seus textos fluentes e ricos em exemplos levam facilmente o leitor por caminhos que de outra forma poderiam parecer extremamente árduos.

Editora Aleph
ALEPH PUBLICAÇÕES E ASSESSORIA PEDAGOGICA LTDA.
CP. 20.707 - CEP. 01498 - SÃO PAULO - [011] 843-3202