LINGUAGEM DE MÁQUINA PARA O TK

TK82, TK83 e TK85

VOL. I - PRINCIPAIS INSTRUÇÕES

FLAVIO ROSSINI

ASSEMBLY Z80

MICRODIGITAL

TK83

Lançamentos
que ajudarão você a descobrir
e explorar todo o potencial
de seu microcomputador.

Dezenas de programas para instrução e lazer em dois volumes.

Curso didático de linguagem Basic para iniciantes, com muitos exemplos e exercícios.

Divirta-se aprendendo truques de programação.

Indicados aos usuários de
TK 82c, TK 83, TK 85, NZ 8000, CP 200,
ZX 81 e TIMEX 1000.

FLAVIO ROSSINI

Alexandre Fernandes
RG: 14.317.733

LINGUAGEM DE MÁQUINA PARA O TK

TK82, TK83 e TK85

VOL. I - PRINCIPAIS INSTRUÇÕES

2ª edição revista

CIP-Brasil. Catalogação-na-Publicação
Câmara Brasileira do Livro, SP

R774L
v. 1 –
2. ed.

Rossini, Flávio, 1959 –
Linguagem de máquina para o TK : TK82, TK83 e TK85 / Flávio Rossini. -- 2. ed. -- São Paulo : Ed. Moderna, 1984 –

Bibliografia.
Conteúdo: v. 1. Principais instruções.

1. Assembly Z80 (Linguagem de programação para computadores) 2. TK (Computador) – Programação I. Título.

84-0258

17. CDD-651.8
18. -001.642
18. -001.6424

Índices para catálogo sistemático:
1. Assembly Z80 : Linguagem de programação : Computadores : Processamento de dados 651.8 (17.) 001.6424 (18.)
2. TK : Computadores : Programação : Processamento de dados 651.8 (17.) 001.642 (18.)

Todos os direitos reservados
EDITORA MODERNA LTDA.
Rua Afonso Brás, 431
Tel.: 531-5099
CEP 04511 - São Paulo - SP - Brasil

1984
Impresso no Brasil

Alexandre Fernandes
RG: 14.317.733

Prefácio

Existem determinadas "frases feitas" quase que obrigatórias quando o assunto é computador, e uma das mais batidas é: "Vivemos uma época de grandes mudanças...". Infelizmente, porém, este tema tem sido muito citado, mas muito pouco aprofundado.

Vivemos uma época de alta tecnologia, mas também de grande miséria. O homem vai à Lua, mas mais da metade da humanidade não saiu da Idade Média. Usamos computadores para apurar resultados de eleições, mas na maioria dos países elas não se realizam de maneira democrática. Um jóquei clube gasta milhões de dólares em equipamentos veterinários para amenizar a dor de um cavalo, e a poucas centenas de metros um ser humano é torturado numa delegacia de polícia.

Qualquer que seja a "cor" política do regime, o Estado é cada vez mais centralizador, onipotente e tirânico. O conflito político de antigamente era entre esquerda e direita. Hoje, é entre o indivíduo e o Estado, seja este comunista, socialista, capitalista ou fascista.

Assim, a grande revolução que o computador inicia não é a tecnológica: é a do indivíduo, sendo o *computador pessoal*, por sua vez, uma arma de defesa do indivíduo contra as pressões impessoais dos grandes grupos e do Estado.

Pela primeira vez na história da Informática posso contrapor o extrato de *minha* conta corrente, elaborado pelo *meu* computador ao do Banco, até então fonte de seu centro de processamento de dados.

Pela primeira vez posso elaborar no *meu* computador um jogo que *eu* ache conveniente para *meus* filhos, sem ser obrigado a comprar um cartucho imbecilizante elaborado por algum eunuco mental de uma multinacional qualquer.

Como toda arma, porém, o computador pessoal tem dois gumes: quem já viu uma escola norte-americana, cheia de crianças robotizadas trabalhando em *silêncio absoluto* em frente a seus computadores, sabe do que estou falando.

Esta total falta de diálogo, este neo-autismo eletrônico abre perspectivas aterrorizantes.

Qual a saída, então? A única que se apresenta é a do conhecimento. Estudar o computador, saber explorar todas as suas reais possibilidades é a única maneira de usá-lo e não se deixar usar por ele. Já ouvi muitas pessoas dizerem: "Comprei

um TK para aprender computação, mas, depois de explorar todas as suas possibilidades, vou partir para um equipamento mais sofisticado". Quando ouço isto tenho vontade de rir e responder: "Moço, se você fosse duas vezes mais inteligente do que é e gastasse dez anos estudando computação em período integral, você não conseguiria explorar nem metade das possibilidades de seu pequeno TK".

As pessoas normalmente vêem apenas a ponta do *iceberg* e nem sequer suspeitam do que existe embaixo.

O livro do Flavio tem o mérito de começar a desvendar a parte escondida do TK, revelando algumas de suas incríveis possibilidades. É um livro escrito de maneira didática, mas não é um livro "fácil". O leitor deve se conscientizar de que este livro deve ser estudado e não simplesmente lido. No fim, porém, os resultados são altamente gratificantes: a linguagem de máquina abre um universo insuspeitado e repleto de possibilidades. Ao dominá-lo é que realmente se sente o milagre que é a posse de uma máquina dessas na própria casa.

Enquanto se discute, em Brasília, a possibilidade de se *fichar* todos os brasileiros, atribuindo *um único número* a cada um, *centralizando* todas as *informações* de um indivíduo no mesmo banco de dados, o Flavio lança este livro. Há uma certa aura de coincidência ver isso acontecer às vésperas de 1984!

Pierluigi Piazzi

Índice

Introdução ... 9

PARTE I — AS INSTRUÇÕES EM LINGUAGEM DE MÁQUINA DO MICROPROCESSADOR Z8Ø

Capítulo 1 — Introdução à linguagem de máquina ... 13
Capítulo 2 — Programando em linguagem de máquina — registros internos do microprocessador, instruções preliminares e sua analogia com o BASIC ... 22
Capítulo 3 — Instruções aritméticas ... 44
Capítulo 4 — Deslocamento de blocos ... 63
Capítulo 5 — O STACK POINTER e o PROGRAM COUNTER: instruções de salto, sub-rotinas e números negativos ... 72
Capítulo 6 — Instruções lógicas e operações com bits ... 97
Capítulo 7 — Finalizando as instruções em linguagem de máquina ... 110

PARTE II — PRIMEIRAS APLICAÇÕES DA LINGUAGEM DE MÁQUINA PARA O TK

Capítulo 8 — Organização de memória do TK ... 125
Capítulo 9 — "Brincando" com a tela de TV ... 144
Capítulo 10 — Entendendo o teclado ... 157
Capítulo 11 — Life: o computador também é artista ... 175

APÊNDICES

Apêndice 1 — Conceitos básicos ... 189
Apêndice 2 — Tabelas das instruções do ASSEMBLY Z8Ø e caracteres do TK 82 (85) ... 198
Apêndice 3 — Love is the answer, and you know that for sure. Love is a flower, you've gotta let it grow ... 239
Bibliografia ... 241

Agradecimentos a

Waléria

Pierluigi

Einstein

e aos Beatles

por todo o amor que torna tudo possível...

INTRODUÇÃO

Este livro foi escrito para pessoas que já estejam familiarizadas com o BASIC-TK, mas que tenham pouco ou nenhum conhecimento sobre linguagem de máquina.

O livro consta de duas partes: a primeira ensina o que é a linguagem de máquina e a segunda apresenta as primeiras aplicações da linguagem no TK, como caráter introdutório ao mundo das possibilidades que esta nova "ferramenta" apresenta. Se você se interessar pelo assunto, encontrará, no segundo volume desta série, aplicações mais aprofundadas dessa linguagem.

Iremos, inicialmente, introduzir o conceito de *microprocessador*. O computador, como nós o conhecemos, consiste nos seguintes elementos principais:

- microprocessador (modelo Z8Ø, no caso do TK);
- memória;
- dispositivos de entrada (teclado, joystick);
- dispositivos de saída (vídeo, impressora);
- memória auxiliar (gravador K7).

O TK pode ser comparado, por analogia, a um ser humano, só que ele "fala" BASIC! Assim, esse "ser" tem um cérebro (que é o microprocessador) e uma memória. Você conversa com ele através do teclado (para enviar informações) e da tela (para receber informações), usando o BASIC. Acontece que o microprocessador *não* entende BASIC... Ele entende uma língua muito mais simples e limitada, à qual chamamos de *linguagem de máquina*. Dessa forma, parte da memória do computador contém um programa escrito em linguagem de máquina capaz de traduzir o BASIC, para que o microprocessador entenda os comandos, e, vice-versa, capaz de traduzir as respostas fornecidas pelo microprocessador.

Nosso intuito será o de conseguir conversar diretamente com o microprocessador na sua própria linguagem! É desejável, para isto, que você esteja bem familiarizado com o funcionamento das instruções PEEK e POKE e com os números escritos em *binário* ou *hexadecimal* (caso você não conheça estes conceitos, leia o apêndice 1 *antes* de ler o primeiro capítulo).

Além disso, começaremos a entender como funciona o programa tradutor, utilizando para nossos programas algumas sub-rotinas já existentes ou valores de algumas variáveis por ele utilizadas para controle do sistema.

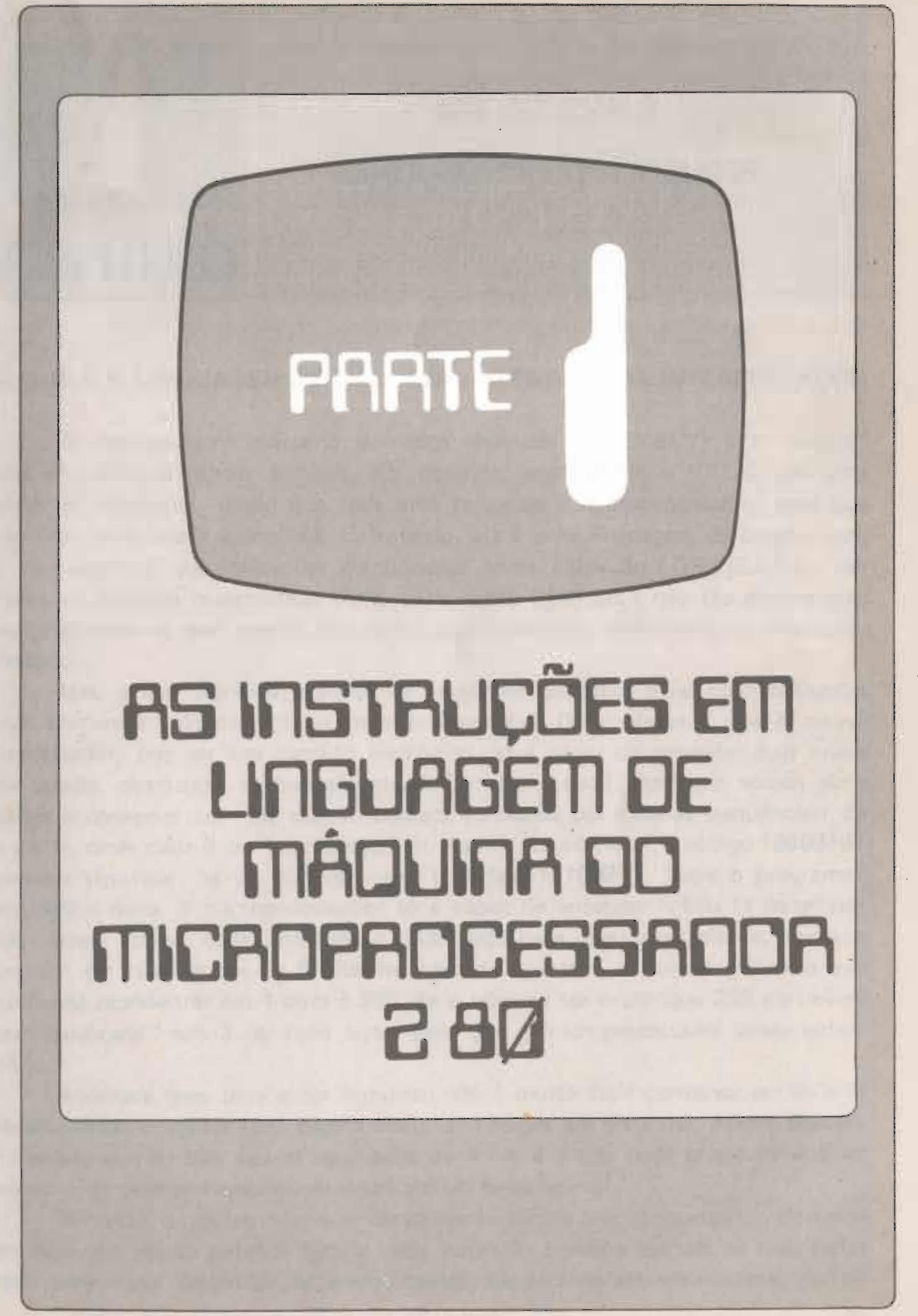
Vários programas serão apresentados e, já na primeira parte deste livro, realizaremos um SCROLL ao contrário, passando a "truques" para lidar com a TV e o teclado e finalizando com um incrível programa chamado LIFE.

Estude com cuidado *cada* programa e procure responder a *todas* as questões que acompanham o texto antes de prosseguir a leitura. No final de cada capítulo haverá exercícios propostos, elaborados não só para recordar os conceitos aprendidos, mas também com o intuito de introduzir outros conceitos importantes e alguns "truques". Portanto, sugerimos a resolução dos exercícios antes de iniciar o capítulo seguinte.

Seria interessante se você tivesse uma expansão de memória de 16K para poder executar os programas mais proveitosos deste livro, embora ela não seja estritamente necessária para aprender a linguagem de máquina.

Você poderia se perguntar: o que esta linguagem tem para me oferecer? A resposta resume-se em duas palavras: *rapidez* e *controle*, e você perceberá isto logo mais!

Bem, seja bem-vindo ao universo da *linguagem de máquina*.



Introdução à Linguagem de Máquina

O QUE É A LINGUAGEM DE MÁQUINA – O PROGRAMA INTERPRETADOR

A linguagem de máquina (também chamada ASSEMBLY) é a "língua" que o microprocessador entende. Ela consiste, assim como o BASIC, em uma série de instruções, sendo que cada uma comanda o microprocessador para que ele faça uma tarefa específica. Entretanto, ela é uma linguagem de baixo nível, o que significa que instruções complicadas como *loops* de FOR/NEXT ou cálculos de funções matemáticas (SIN, COS, EXP, RND etc.) não são disponíveis; ou seja, pode-se usar apenas instruções que comandam diretamente o microprocessador.

Mas, o que significa exatamente linguagem de baixo nível com instruções que comandam diretamente o microprocessador? O problema é que o microprocessador, por ser um circuito eletrônico, só é capaz de entender dois níveis de tensão, chamados, simbolicamente, Ø (zero) e 1 (um). Portanto, somos obrigados a conversar com ele usando códigos formados por cadeias (seqüências) de Øs e 1s, onde cada Ø ou 1 é chamado *bit*. Assim, por exemplo, o código 1ØØØØ1Ø1 poderia significar "some dois números; o código 1Ø1ØØØ11, "pare o programa" etc. Além disso, o microprocessador só é capaz de entender 8 bits (1 *byte*) por vez! Dessa forma, cada instrução e cada dado deve, obrigatoriamente, consistir em um ou mais grupos de 8 bits. No caso de números, o número máximo que podemos representar em 1 byte é 255. Se o número for maior que 255, ele deverá ser "quebrado" em 2 ou mais bytes para que o microprocessador possa entendê-lo.

Acontece que, para o ser humano, não é muito fácil conversar em Øs e 1s (basta tentar imaginar uma página cheia de códigos em Øs e 1s). Assim, convencionou-se que os bits seriam agrupados de 4 em 4 e que cada grupo deveria ser substituído pelo correspondente algarismo em hexadecimal.

Portanto, se quisermos falar diretamente com o microprocessador, devemos lembrar que ele só entende bits, e cada instrução consiste em um ou mais bytes que, para maior facilidade de entendimento, são escritos em hexadecimal, corres-

pondendo, então, cada byte a dois dígitos hexadecimais (por exemplo, C9, D4, 37 etc.).

Observação: Para quem não está familiarizado com as notações indicadas, sugerimos novamente que, antes de prosseguir, leia o apêndice 1: números binários e hexadecimais, bit e byte, PEEK e POKE.

A MEMÓRIA DO COMPUTADOR: ROM E RAM

Assim como as instruções são sempre divididas em grupos de 8 bits, a memória do computador também é divida em regiões capazes de armazenar apenas 8 bits cada uma.

ENDEREÇO DECIMAL	ENDEREÇO HEXADECIMAL	CONTEÚDO DA MEMÓRIA								CONTEÚDO EM HEXADECIMAL
0	ØØØØ	Ø	1	1	Ø	1	1	1	1	6F
1	ØØØ1	1	Ø	Ø	1	Ø	1	1	1	97
2	ØØØ2	1	Ø	1	Ø	1	Ø	1	1	AB
3	ØØØ3	Ø	Ø	1	1	Ø	1	Ø	1	35
4	ØØØ4	1	1	1	Ø	1	1	Ø	1	ED
5	ØØØ5	Ø	1	Ø	Ø	1	1	1	Ø	4E
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
65530	FFFA	1	1	Ø	Ø	Ø	Ø	Ø	1	C1
65531	FFFB	Ø	1	Ø	Ø	Ø	Ø	Ø	Ø	4Ø
65532	FFFC	Ø	1	Ø	Ø	Ø	Ø	1	Ø	42
65533	FFFD	1	1	1	1	Ø	1	Ø	1	F5
65534	FFFE	Ø	Ø	1	1	1	Ø	1	Ø	3A
65535	FFFF	1	Ø	1	1	1	Ø	Ø	1	B9

Fig. 1.1 – Exemplo de como pode ser imaginada uma memória com seus respectivos endereços e conteúdo. Note que o conteúdo de um registro de memória tem sempre 1 byte (8 bits), podendo ser representado com 2 dígitos hexadecimais, e seu respectivo endereço 2 bytes (16 bits), podendo ser representado com 4 dígitos hexadecimais.

A memória pode ser imaginada como uma pilha de registros capazes de armazenar apenas 1 byte cada, sendo que cada registro tem um endereço que, no caso do TK, pode ir de Ø até 65535 (correspondendo a ØØØØ até FFFF em hexadecimal). (Ver figura 1.1.)

A memória é basicamente dividida em duas partes: a primeira, chamada ROM (Read Only Memory), foi previamente gravada por processos especiais e contém o programa que *interpreta* a linguagem BASIC, ou seja, *traduz* as instruções do BASIC para a linguagem de máquina. Esta região da memória é inalterável, não podendo, portanto, ser escrita (apenas lida), e não perde nunca seu conteúdo, mesmo desligando-se o computador. A segunda, chamada RAM (Random Access Memory), é alterável (pode ser lida ou escrita), e é onde são armazenados o conteúdo da tela de TV, os programas em BASIC devidamente traduzidos pelo interpretador (ROM), as variáveis dos programas e, inclusive, os futuros programas em linguagem de máquina que faremos neste livro. Ela perde seu conteúdo quando o computador é desligado.

Note que o programa interpretador (ou tradutor) que está na memória ROM é escrito em linguagem de máquina, pois o microprocessador deve ser capaz de entendê-lo. Entretanto, como todo programa, ele possui variáveis, que devem ser, obrigatoriamente, colocadas na RAM (caso contrário não seriam variáveis). Assim, ele reserva para si o início da memória RAM, mais precisamente os endereços 16384 a 165Ø8, para colocar suas próprias variáveis.

Portanto, os programas que fizermos em linguagem de máquina deverão ser colocados em alguma região da memória RAM, inteligentemente escolhida, para que não afetem o andamento dos demais programas (eventuais programas em BASIC), a própria tela de TV ou as variáveis do programa interpretador. Quando programamos em BASIC, estes cuidados não são necessários pois o programa residente na ROM se encarrega de colocar as instruções em lugares adequados da RAM.

Veremos agora *como* e *onde* podemos colocar um programa em linguagem de máquina no computador.

O PROGRAMA HEXAMEM – ONDE COLOCAR OS PROGRAMAS EM LINGUAGEM DE MÁQUINA

Um bom lugar para colocar os programas em linguagem de máquina é no fim da memória RAM pois, como já dissemos, o início da memória RAM é reservado para colocar as variáveis do programa interpretador, o programa em BASIC e suas variáveis e a tela de TV. (Ver figura 1.2.)

Entre as variáveis do programa interpretador está uma, chamada RAMTOP, que indica o fim da memória do computador, ou seja, nos registros da memória correspondentes a essa variável o programa interpretador coloca o endereço que

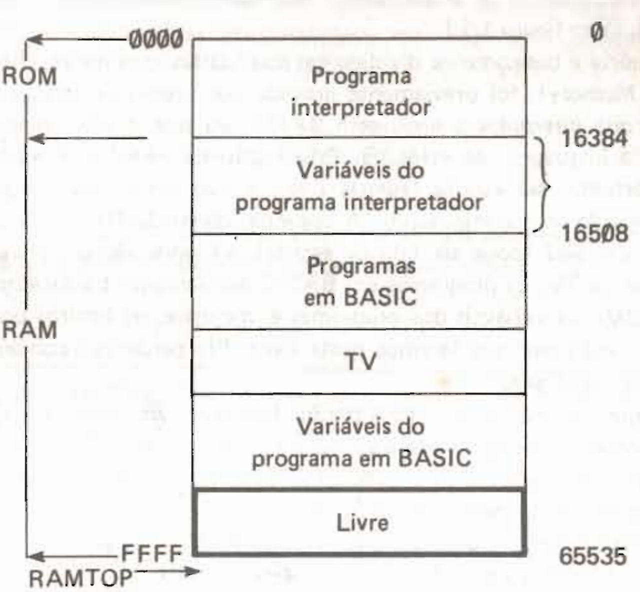


Fig. 1.2 – A memória do TK. A memória só irá até 65535 se usarmos a expansão de 64K.

seria do byte imediatamente após o último byte de memória. Esta variável está colocada nos endereços 16388 e 16389 pois, sendo ela um endereço, tem 16 bits, devendo então ser "quebrada" em duas partes para poder ser colocada na memória, armazenando-se antes o byte *menos* significativo (esta regra é usada para todos os dados de 16 bits!).

Assim, digite (N.L. = NEW LINE):

PRINT PEEK 16388 + (256 * PEEK 16389) (N.L.)

e você obterá o valor de RAMTOP que dependerá de quanta memória você tem disponível. (Consulte o apêndice 1, em caso de dúvida.) Se você tiver a expansão de 16K (suposição válida para todo o livro), deverá obter

RAMTOP = 32768 ('8000' em hexadecimal)

o que significa que o endereço do último byte da memória é 32767 ('7FFF' em hexadecimal).

Vamos, então, "enganar" o computador e modificar o valor desta variável, com o intuito de reservar o fim da memória RAM para os programas em linguagem de máquina, sem prejudicar o resto da memória.

Por exemplo, façamos RAMTOP = 30000 ('7530' em hexadecimal), lembrando que devemos colocar antes o byte menos significativo ('30'). Digite:

POKE 16388,48 (N.L.) (48 = '30' em hexadecimal)
POKE 16389,117 (N.L.) (117 = '75' em hexadecimal)
NEW (N.L.)

o que corresponde a fazer

Alexandre Fernandes
RG: 14.317.733

RAMTOP = 256 * 117 + 48 = 30000

(verifique isto usando novamente a função PEEK para obter o valor de RAMTOP).

Desse modo você reservou a memória 30000 até 32767 para colocar seus programas em linguagem de máquina, pois os demais programas contidos no computador, suas variáveis e o conteúdo da tela de TV *nunca* invadirão esta região por "pensar" que a memória termina no endereço 29999 ('752F' em hexadecimal). Note, todavia, que esta região *não* será afetada pelo comando NEW e *não* poderá ser passada para a fita através do comando SAVE. Esta última limitação é desagradável, porém mais adiante veremos como contorná-la.

Observação: Caso você não tenha expansão de memória, use:

POKE 16388,173
POKE 16389,67
NEW

para reservar a região de 17325 a 18431.

Apresentaremos agora um programa em BASIC que coloca na memória RAM um programa em linguagem de máquina elaborado em códigos hexadecimais a partir de um endereço inicial fornecido por você (lembre-se de reservar espaço no fim da memória antes de colocar o programa):

POKE 16388,48 (N.L.)
POKE 16389,117 (N.L.)
NEW (N.L.)

A seguir, coloque HEXAMEM no computador e introduza alguns códigos hexadecimais (que por enquanto nada significam) dentro do computador. Cada código deve ser seguido de NEW LINE (N.L.), e o programa termina ao introduzir-se a letra *P* (pare).

```
  0 REM FLAVIO ROSSINI
  5 REM HEXAMEM
 10 FAST
 25 PRINT "ESCOLHA O ENDERECO I
NICIAL DA"
 30 PRINT "MEMORIA     (>=30000)"
 35 INPUT IN
 45 PRINT "MEMORIA INICIAL = ";
IN
 50 LET INI=IN
 55 LET A$=""
 60 SCROLL
 65 PRINT "MEM. ";IN;
 70 IF A$="" THEN INPUT A$
 75 IF A$="P" THEN STOP
 80 IF A$="XS" THEN GOTO 0130
 85 IF A$="XF" THEN GOTO 0135
 90 LET AUX=16*CODE A$+CODE A$(
2)-476
 95 LET X$=STR$ (16*CODE A$+COD
E A$(2)-476)
100 GOSUB 0200
105 PRINT TAB 13;C$;B$;TAB 24;A
$( TO 2);TAB (31-LEN X$);X$
110 POKE IN,AUX
115 LET IN=IN+1
120 LET A$=A$(3 TO )
125 GOTO 0060
130 SLOW
135 CLS
140 PRINT AT 3,0;USR INI
145 STOP
200 LET I=1
205 GOSUB 0500
210 LET I=2
215 LET C$=B$
220 GOSUB 0500
225 RETURN
500 IF A$(I)="0" THEN LET B$="0
000"
505 IF A$(I)="1" THEN LET B$="0
301"
510 IF A$(I)="2" THEN LET B$="0
010"
515 IF A$(I)="3" THEN LET B$="0
011"
520 IF A$(I)="4" THEN LET B$="0
100"
525 IF A$(I)="5" THEN LET B$="0
101"
530 IF A$(I)="6" THEN LET B$="0
110"
535 IF A$(I)="7" THEN LET B$="0
111"
540 IF A$(I)="8" THEN LET B$="1
000"
545 IF A$(I)="9" THEN LET B$="1
001"
550 IF A$(I)="A" THEN LET B$="1
010"
555 IF A$(I)="B" THEN LET B$="1
011"
560 IF A$(I)="C" THEN LET B$="1
100"
565 IF A$(I)="D" THEN LET B$="1
101"
570 IF A$(I)="E" THEN LET B$="1
110"
575 IF A$(I)="F" THEN LET B$="1
111"
580 RETURN
```

Prog. 1.1 – HEXAMEM.

Observação: Se você não tiver expansão de memória, substitua a linha 3Ø por

PRINT "MEMÓRIA (>= 17325)"

e retire as linhas 95, 1ØØ e de 2ØØ a 58Ø, substituindo a linha 1Ø5 por

PRINT TAB 13; A$ (TO 2); TAB 17; AUX

Este programa fornece, também, uma visualização da memória do computador na tela de TV da seguinte maneira:

MEM.	N.DA MEM. EM DECIM.	CONTEÚDO DA MEM. EM BINÁRIO	CONTEÚDO DA MEM. EM HEXADECIMAL	CONTEÚDO DA MEM. EM DECIMAL

Experimente, então, colocar alguns números hexadecimais na memória, a partir do endereço 3ØØØØ. Lembre-se de que cada endereço da memória corresponde a 1 byte de dados. Portanto, você deve entrar sempre com um número par de dígitos hexadecimais; caso contrário, o programa parará na linha 90. Assim, entre, por exemplo, com o seguinte:

RUN	(N.L.)
Memória inicial: 3ØØØØ	(N.L.)
A4	(N.L.)
B7Ø412	(N.L.)
Ø6	(N.L.)
71FDA1	(N.L.)
ØØCE	(N.L.)
P	(N.L.)

Na tela, você deverá obter:

```
MEM. 30000   10100100   A4  164
MEM. 30001   10110111   B7  183
MEM. 30002   00000100   04    4
MEM. 30003   00010010   12   18
MEM. 30004   00000110   06    6
MEM. 30005   01110001   71  113
MEM. 30006   11111101   FD  253
MEM. 30007   10100001   A1  161
MEM. 30008   00000000   00    0
MEM. 30009   11001110   CE  206
MEM. 30010
```

Com isto, você colocou estes dados da memória 3ØØØØ até a memória 3ØØØ9. Vamos verificar se o programa funcionou. Acrescente, agora, o seguinte programa:

```
3000 REM VERMEM
3005 SLOW
3010 PRINT "FIM = ?"
3015 INPUT FIM
3020 FOR I=30000 TO FIM
3025 SCROLL
3030 PRINT "MEMORIA",I;"      ";PE
EK I
3035 NEXT I
```

Prog. 1.2 – VERMEM.

Observação: Se você não tiver expansão de memória, substitua a linha 3020 por:

FOR I = 17325 TO FIM

Execute-o fazendo RUN 3000 (ou GOTO 3000).

Coloque, para a variável FIM, o valor 30009 (17334 sem expansão), pois colocamos 10 códigos hexadecimais na memória. Ao executar o programa, você deverá obter os números previamente colocados, mas em decimal! Portanto, teremos:

MEM. 30000	164	(= 'A4')
MEM. 30001	183	(= 'B7')
MEM. 30002	4	(= '04')
MEM. 30003	18	(= '12')
MEM. 30004	6	(= '06')
MEM. 30005	113	(= '71')
MEM. 30006	253	(= 'FD')
MEM. 30007	161	(= 'A1')
MEM. 30008	0	(= '00')
MEM. 30009	206	(= 'CE')

Representações em hexadecimal (veja os códigos previamente colocados).

repita este procedimento para outros números hexadecimais até entender o que está acontecendo...

Note que estamos apenas colocando os códigos hexadecimais na memória, e, mesmo que esses códigos (A4, B7, 04, 12 etc.) significassem um programa em linguagem de máquina, providências especiais seriam necessárias para executar o programa. Estas providências serão vistas no próximo capítulo.

Estes códigos podem representar um programa em linguagem de máquina, colocado a partir da memória 30000. Por exemplo, o código 164 (A4 em hexadecimal) poderia significar "some os próximos dois bytes (183 e 4)"; o código 18 (12 em hexadecimal), "compare o resultado com o próximo byte (6)" e assim por diante...

Note que não existem endereços para as linhas de programa como em BASIC, e que as instruções são executadas na seqüência em que foram colocadas na memória.

Com todos esses "poréns", quais seriam, então, as vantagens da linguagem de máquina com relação ao BASIC?

A linguagem de máquina é *muito mais rápida* e permite uma *visualização* e um *controle* maior do sistema. Em contrapartida, suas instruções são mais simples e limitadas e exigem um cuidado maior na hora da programação.

Resumo

Vimos, neste capítulo, como são as instruções em linguagem de máquina (sempre em grupos de dois dígitos hexadecimais), o que é memória ROM e RAM e como reservar espaço no fim da memória RAM para colocar lá o programa em linguagem de máquina.

Foi apresentado um programa em BASIC (HEXAMEM) que permite colocar programas elaborados em códigos hexadecimais no fim da memória RAM a partir de um endereço maior ou igual a 30000. Para isso modificamos o valor de uma variável do programa interpretador, chamada RAMTOP, cujos endereços são 16388 e 16389 (pois ela tem 2 bytes) e que indica o fim da memória do computador.

Exercícios

1. Analise o funcionamento do programa HEXAMEM, exceto as linhas 130, 135 e 140.
2. Analise o funcionamento do programa VERMEM.

Programado em Linguagem de Máquina

Registros Internos do Microprocessador

Instruções Preliminares e sua Analogia com o BASIC

PRINCIPAIS REGISTROS INTERNOS DE 8 BITS, OS MNEMÔNICOS E A FUNÇÃO USR

Iniciaremos o capítulo explorando o conceito de *registro*, termo empregado propositadamente no 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 Ø a 255 ('ØØ' a 'FF' em hexadecimal). (Ver apêndice 1.)

O microprocessador do TK (Z8Ø) 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 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 e HL (só estes pares são possíveis).

Por exemplo, se o registro B contém o número '1A' (16Ø em decimal) e o registro C contém 'B2' (178 em decimal), o par BC contém o número '1AB2' (256 * 16Ø + 178 = 41138 em decimal).

µp

A	
B	C
D	E
H	L

Fig. 2.1 — Como podem ser imaginados os principais registros internos do microprocessador (µp).

Como vimos no capítulo anterior, as instruções em linguagem de máquina são representadas por códigos em hexadecimal de dois dígitos cada. Isto é feito para evitar a confusão que poderia ser criada usando apenas os números Ø e 1. Mesmo assim, temos agora apenas 16 símbolos disponíveis (Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, 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, chamados *códigos de operação*, que têm como objetivo associar uma *ordem* a cada grupo de letras. Por exemplo, se o código 1Ø1ØØ111 significar "some", seu correspondente em hexadecimal seria 'A7' 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, substituiremos os mnemônicos por seus códigos hexadecimais e utilizaremos o HEXAMEM 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.

No TK 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) cujo argumento é o endereço inicial da sub-rotina (por exemplo: 3ØØØØ se o programa em linguagem de máquina estiver a partir do endereço que colocamos em RAMTOP no capítulo 1); portanto, estas sub-rotinas devem ter como instrução final um comando para voltar ao BASIC, similar ao RETURN, cujos código hexadecimal e mnemônico são, respectivamente:

'C9' (código)   RET (mnemônico abreviação de RETurn)

Vamos entender como funciona a instrução USR (instrução do BASIC do tipo FUNÇÃO). Suponhamos que tivéssemos o seguinte programa em BASIC:

```
1Ø PRINT USR 3ØØØØ
```

Ao encontrar USR 3ØØØØ, o computador "carrega" o par de registros BC (mencionados anteriormente) com este número, ou seja, como 3ØØØØ = '753Ø', teremos B = '75' e C = '3Ø'. A seguir, o microprocessador executa o programa em linguagem de máquina que começa na memória 3ØØØØ (= '753Ø'); portanto, é conveniente que tenha sido previamente colocado um programa a partir da memória 3ØØØØ! O programa é executado até encontrar a instrução RET (código 'C9'), quando, então, o controle volta ao programa BASIC fornecendo como saída o conteúdo do par BC, que, se não for alterado durante o programa em

linguagem de máquina, continuará sendo '7530' (30000). Portanto, para obter a saída da sub-rotina (ou seja, os resultados que devem estar no par BC) basta ordenar um PRINT da função USR.

Vamos então colocar o seguinte programa em linguagem de máquina na memória 30000 (que você já deve ter reservado com os POKEs do capítulo 1):

RET

(apenas isto!).

Como se trata apenas de uma instrução, não utilizaremos o programa HEXAMEM do capítulo 1, mas sim a instrução POKE. O código hexadecimal do RET é 'C9', que equivale a 201 (em decimal); portanto, digite:

POKE 30000, 201 (N.L.)

A seguir digite:

PRINT USR 30000 (N.L.)

O que deverá aparecer na tela? Ora, a função USR carregará o par BC com 30000 ('7530'), e ordenará a execução da sub-rotina a partir da memória 30000, encontrando imediatamente o RET. Com isso o controle voltará ao programa em BASIC sem ter alterado os registros BC e fornecerá como saída o próprio 30000; portanto, na tela de TV deverá aparecer o número 30000.

A instrução RET é necessária pois, ao contrário do BASIC, o programa ao ser executado *não* pára automaticamente na última instrução. Ele continuará tentando interpretar as próximas regiões de memória que, simplesmente, não terão sentido, e "coisas estranhas" poderão ocorrer... O mesmo é válido se houver algum erro drástico no programa, podendo fazer a tela "sumir" ou produzir algum belo quadro de arte moderna; e você perceberá que provavelmente nem a tecla BREAK será capaz de corrigir isto! Portanto, a única maneira de se fazer tudo voltar ao normal é desligar o computador, o que causará a perda do programa.

Poderia surgir agora a seguinte pergunta: mas não existe uma instrução equivalente ao STOP? De fato, há em linguagem de máquina uma instrução, similar ao STOP em BASIC, cujo código de operação é HALT (código hexadecimal '76'). No entanto, sugerimos que você *nunca* use esta instrução pois, *neste* computador, ela paralisa o sistema e este não aceita mais nenhuma tecla, nem mesmo o BREAK. Portanto, use sempre RET para terminar seus programas em linguagem de máquina.

Vamos agora começar a "brincar" um pouco com os registros.

A INSTRUÇÃO LD: COLOCAÇÃO DE NÚMEROS NOS REGISTROS E CÓPIA DE UM REGISTRO EM OUTRO

A instrução LD permite colocar números dentro dos registros internos do

microprocessador (obviamente estes números poderão ser de '00' a 'FF' 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 próprio código hexadecimal. (LD é abreviação de LoaD = carregue.)

Vamos analisar inicialmente como colocar números diretamente nos vários registros; a instrução, de modo genérico, corresponde a:

LD registro,dado

Por exemplo:

LD E, '2A'

(que equivale a dizer LD E, 42), colocará o número '2A' (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:

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

Tab. 2.1 — A instrução LD registro,dado (8 bits).

Faremos então um programa que carrega o registro B com '00' e o registro C com '2A':

```
MEM. 30000    LD B,'00'    '0600'   ;'00'= 0
MEM. 30002    LD C,'2A'    '0E2A'   ;'2A' = 42
MEM. 30004    RET          'C9'
```

Prog. 2.1 — Programa de 5 bytes que carrega o par BC com o número 42 ('002A').

Podemos então colocá-lo, por exemplo, a partir da memória 30000 (lembre-se de reservá-la), usando cinco vezes POKE como fizemos anteriormente para a instrução RET (transformando adequadamente os códigos hexadecimais em decimais) ou, então, usando o programa HEXAMEM para introduzir as três instruções.

Se você utilizar o HEXAMEM, lembre-se de reservar o fim da memória *antes* de colocá-lo no computador; a seguir, execute-o colocando 30000 para o endereço inicial e faça:

0600 (N.L.)
0E2A (N.L.)
C9 (N.L.)
P (N.L.)

Note que o programa ocupa cinco posições de memória, de 30000 a 30004, respectivamente.

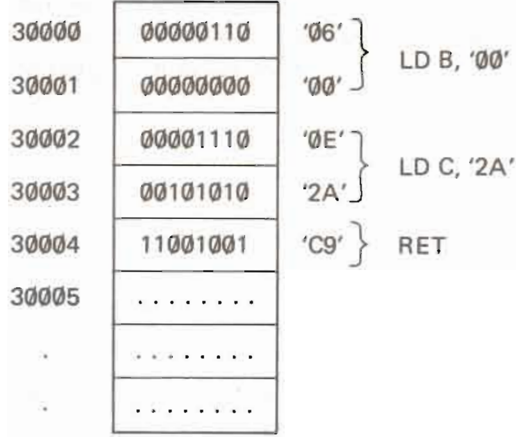


Fig. 2.2 – Visualização do programa na memória do computador.

A seguir, digite:

PRINT USR 30000 (N.L.)

O que deverá aparecer na tela? O conteúdo do par de registros BC, ou seja, 256 * 0 + 42 = 42 (isto porque o par de registros foi alterado antes da instrução RET). Assim, seu conteúdo inicial, que era BC = 30000 ('7530'), foi modificado pelo programa para BC = 42 ('002A').

Há também a possibilidade de carregar diretamente pares de registros; neste caso, porém, cada instrução corresponderá a 3 bytes, 1 para o código de operação e 2 para o número que deverá ter 16 bits. Note que, como já foi comentado anteriormente, você deverá sempre colocar antes o byte *menos* significativo. Assim, temos as seguintes instruções:

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

Tab. 2.2 – A instrução LD par de registros,dado (16 bits).

Façamos o programa:

MEM. 30000	LD BC, '002B'	'012B00'	; note a inversão: '2B' é colocado antes de '00'
MEM. 30003	RET	'C9'	

Prog. 2.2 – Programa de 4 bytes para carregar diretamente o par BC com 43.

Usando HEXAMEM, coloque o programa a partir da memória 30000. Verifique que fazendo PRINT USR 30000 você obterá 43 ('002B').

Observação: Você não precisa se preocupar em "apagar" o programa anterior pois estará escrevendo "por cima" nas mesmas memórias.

Experimente "esquecer" de inverter os bytes, conforme indicado, e introduza este programa:

```
'01002B'
'C9'
```

Prog. 2.3 – Programa 2.2 sem a inversão.

O que você irá obter? Vejamos: 256 * 43 + 0 = 11008, ao invés de 43. Perceba então como é importante lembrar da *inversão* para números que ocupam *2 bytes*.

Veremos agora como passar números de um registro para outro. Temos todas as combinações possíveis da seguinte instrução genérica:

LD registro,registro

que copia o conteúdo do registro da direita no registro da esquerda, *não* alterando o conteúdo do registro da direita; essa instrução equivale a apenas 1 byte em código hexadecimal pois não exige que nenhum dado seja explicitado. Para facilitar, construiremos uma tabela que possui todas as combinações possíveis dessa instrução.

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

Tab. 2.3 – Instrução: LD registro,registro.

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

LD C,D equivale a '4A'

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

LD E,E equivale a '5B'

(copie o conteúdo do registro E no registro E. Como você pode notar, essa instrução 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 (lembre-se que, por enquanto, iremos fazer sempre MEMÓRIA INICIAL = 30000, e que o final de memória *deve* estar reservado antes de colocar HEXAMEM no computador):

```
MEM. 30000   LD H,'00'   '2600'   ; carrega H com '00'
MEM. 30002   LD E,'2A'   '1E2A'   ; carrega E com '2A'
MEM. 30004   LD B,H      '44'     ; copia H em B
MEM. 30005   LD C,E      '4B'     ; copia E em C
MEM. 30006   RET         'C9'     ; volta ao BASIC
```

Prog. 2.4 – Programa de 7 bytes para copiar registros.

Utilize então HEXAMEM para colocar os códigos do programa na memória e, no final, ao invés de teclar *P* (pare), tecle XS (eXecute em Slow) ou XF (eXecute em Fast) para que o programa seja executado. Naturalmente, se você prefere continuar fazendo PRINT USR 30000 não há problemas.

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

Observação: Não existe instrução para copiar de uma só vez números de um par de registros para outro; assim, 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 instruções 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.

É conveniente lembrar que ao chamar uma sub-rotina em linguagem de máquina, só temos certeza do conteúdo dos registros B e C; todos os demais têm um conteúdo desconhecido. Não existe uma instrução análoga ao CLEAR para "zerar" estes registros, e assim os seus conteúdos inicias são "aleatórios". O mesmo é válido para a memória que fica *após* o programa. Vamos exemplificar: ao fazer RAMTOP = 30000, reservamos a memória de 30000 a 32767. Se tivermos um programa que ocupe 100 bytes, ou seja, da memória 30000 à memória 30099, as posições de 30100 a 32767 terão valores "quaisquer", e é por isto que o microprocessador fica "maluco" se esquecermos do RET. É nesta região de memória, *após* o programa em linguagem de máquina, que colocaremos as variáveis do nosso programa que não "couberem" nos registros.

A INSTRUÇÃO LD PARA ENDEREÇAMENTO DIRETO E INDIRETO: CÓPIA DE REGISTROS NA MEMÓRIA E VICE-VERSA

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 à disposição toda a memória RAM. Naturalmente, devemos tomar cuidado para não interferirmos com os demais programas ou com a tela de TV; por enquanto este problema não nos atinge pois estamos utilizando uma região de memória reservada no fim da RAM.

Se você escrever, em BASIC:

```
LET X = PEEK 3ØØØØ
```

o que isto significa?

Ora, a variável X irá assumir um valor entre Ø e 255, que corresponde ao que está na memória 3ØØØØ. Em linguagem de máquina existe uma instrução análoga, mas que funciona apenas com o registro A:

```
LD A, (3ØØØØ)
```

ou

```
LD A, ('753Ø')
```

cujo código é '3A' + 2 bytes para o endereço.

Note que os parênteses indicam a seguinte idéia: copie no registro A o conteúdo da memória 3ØØØØ. De fato, a instrução LD A, 3ØØØØ (sem parênteses) não é válida porque não podemos colocar o número 3ØØØØ dentro de um único registro (esse número é maior que 255). Esse tipo de instrução é chamado endereçamento *direto*, pois podemos dizer *diretamente* na instrução em que endereço da memória está o que queremos colocar no registro A.

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

Experimente, então, colocar um número qualquer na memória: por exemplo, 162 na memória 3ØØØ7. Digite:

```
POKE 3ØØØ7,162     (N.L.)
```

e execute o seguinte programa:

```
MEM. 3ØØØØ  LD A, (3ØØØ7)   '3A3775'   ; (pois 3ØØØ7 = '7537')
MEM. 3ØØØ3  LD B, Ø         'Ø6ØØ'     ; coloca Ø em B
MEM. 3ØØØ5  LD C, A         '4F'       ; coloca A em C
MEM. 3ØØØ6  RET             'C9'       ; volta ao BASIC
```

Prog. 2.5 – Programa de 7 bytes para copiar o conteúdo da memória 3ØØØ7 usando endereçamento direto.

Essa sub-rotina copia o conteúdo da memória 3ØØØ7 no registro A e carrega o par BC com este número (note que fazemos B = Ø); portanto, na tela deverá aparecer o número 162. O que aconteceria se retirássemos a instrução LD B, Ø? Note que o valor de B continuaria sendo '75' pois a sub-rotina, ao ser chamada, carregou BC com '753Ø' (3ØØØØ). Dessa forma, obteríamos:

117 * 256 + 162 = 3Ø114 ('75' = 117)

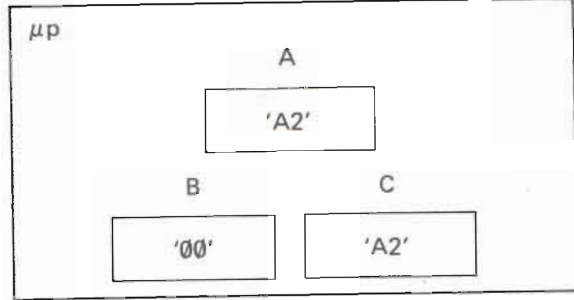


Fig. 2.3 – Visualização do microprocessador.

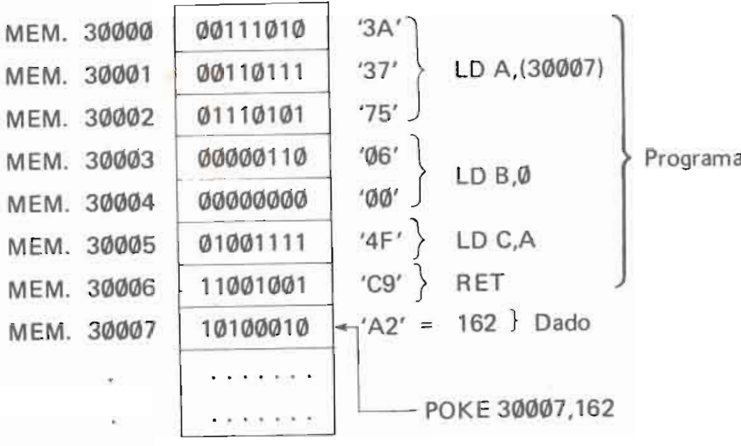


Fig. 2.4 – Visualização da memória.

Observação: Colocamos o número 162 na memória 3ØØØ7 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 memórias ocupadas 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 Ø a 65535)? De fato, em linguagem de máquina é possível a instrução do tipo

LD registro,(par de registros)

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; assim, temos:

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

Tab. 2.4 — A instrução LD registro(par de registros).

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, E, H, 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).

Coloque novamente um número na memória 3ØØØ7:

POKE 3ØØØ7, 25Ø

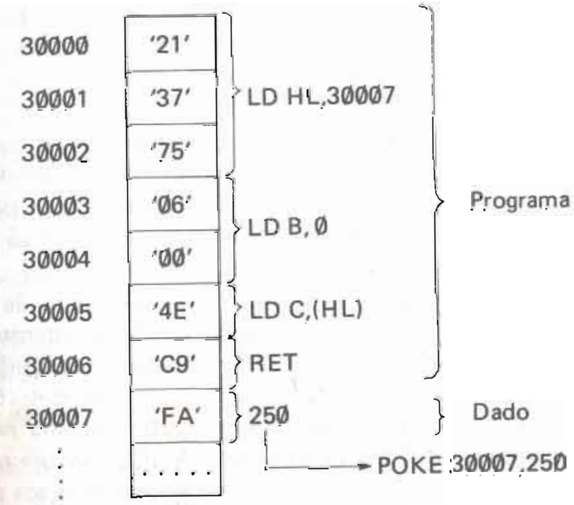
e execute (você deverá obter 25Ø na tela):

MEM. 3ØØØØ	LD HL,3ØØØ7	'213775'	; (3ØØØ7 = '7537') coloca o número 3ØØØ7 no par HL
MEM. 3ØØØ3	LD B,Ø	'Ø6ØØ'	; coloca Ø em B
MEM. 3ØØØ5	LD C,(HL)	'4E'	; copia em C o conteúdo da memória apontada por HL (3ØØØ7)
MEM. 3ØØØ6	RET	'C9'	

Prog. 2.6 — Programa de 7 bytes para copiar o conteúdo da memória 3ØØØ7 usando endereçamento indireto.

(Observe que novamente colocamos com POKE o número diretamente *após* o fim do programa.)

Esse programa carrega o par HL com 3ØØØ7 e, a seguir, coloca em C o conteúdo da memória que está endereçada por HL. Note que precisamos fazer B = Ø pois o conteúdo da memória 3ØØØ7 tem apenas 8 bits e ocupará somente o registro C.



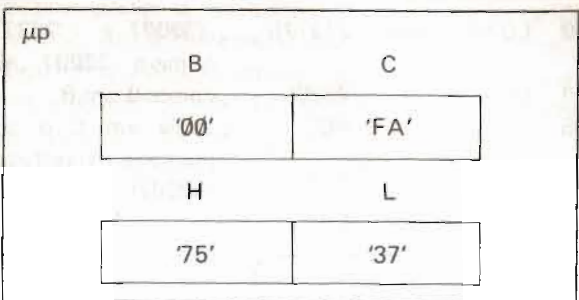


Fig. 2.5 – Visualização do microprocessador e da memória.

Assim como o PEEK tem sua função inversa, o POKE, as instruções que acabamos de ver também têm suas funções inversas, a saber:

INSTRUÇÃO	CÓDIGO
LD (endereço),A	'32' (+2 bytes de endereço)
LD (BC),A	'02'
LD (DE),A	'12'
LD (HL),A	'77'
LD (HL),B	'70'
LD (HL),C	'71'
LD (HL),D	'72'
LD (HL),E	'73'
LD (HL),H	'74'
LD (HL),L	'75'

Tab. 2.5 – Instruções para colocar o conteúdo dos registros na memória.

Note que a primeira instrução desta lista corresponde a 3 bytes enquanto as demais apenas a um. 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.

Faremos mais um programa para exemplificar o que acabamos de aprender: no capítulo 1 fizemos referência às variáveis do *programa interpretador*, que estão no início da memória RAM (entre 16384 e 16508), e exemplificamos dizendo que a variável RAMTOP tinha 2 bytes e estava *nos endereços* 16388 e 16389. Vamos agora utilizar *outra* variável de apenas 1 byte, que está no endereço 16442. Ela indica a posição da linha PRINT (em BASIC), com um número de 1 a 24 pois, como sabemos, a tela do TK tem 22 linhas acessíveis aos programas BASIC mais duas linhas de edição. É considerada a linha 1 aquela "mais baixa" da tela; experimente executar (RUN 4000):


```
4000 SLOW
4005 FOR I=0 TO 21
4015 PRINT PEEK 16442
4025 NEXT I
```

Prog. 2.7 – A utilização da variável que indica a linha de PRINT.

Este programa deve esclarecer o que significa essa variável. Caso seu significado ainda tenha ficado duvidoso, tente complementar o programa da seguinte maneira:

```
4000 SLOW
4005 FOR I=0 TO 21
4010 FOR J=0 TO 3
4015 PRINT TAB (8*J);PEEK 16442;
4020 NEXT J
4025 NEXT I
```

Prog. 2.8 – Reutilização da variável que indica a linha de PRINT.

Vamos então simular este PEEK com uma sub-rotina em linguagem de máquina (digite *P* no final do programa):

```
MEM. 30000 LD A,(16442)   '3A3A40'  ; (16442 = 403A)
MEM. 30003 LD B,0         '0600'
MEM. 30005 LD C,A         '4F'
MEM. 30006 RET            'C9'
```

Prog. 2.9 – Simulação de um PEEK.

Essa sub-rotina carrega o par de registros BC com o conteúdo da memória 16442. Modifique a linha 4015 do programa BASIC:

4015 PRINT TAB (8*J); USR 30000;

Execute o programa e você deverá obter o mesmo resultado anterior. Na realidade, essa sub-rotina faz exatamente o mesmo que a função PEEK: coloca

no acumulador o conteúdo da memória 16442 e, para obter o resultado na tela, copia o conteúdo do acumulador em C, fazendo B = Ø.

Na memória 16441 existe outra variável de 1 byte que indica o número da coluna para PRINT. Experimente fazer um programa similar ao 2.8 mas que use uma sub-rotina em linguagem de máquina para realizar o PEEK 16441.

O que acontecerá ao programa se você esquecer de colocar a instrução LD B,Ø? Experimente executar qualquer um dos três últimos programas omitindo esta instrução; você obterá como resultado a resposta acrescida de 29952. Veja porque: inicialmente, ao aparecer USR 3ØØØØ, o par de registros BC é carregado com 3ØØØØ, mas você altera apenas o registro C, pois seus resultados são menores que 255. O registro B continua com seu valor inicial, 117, e assim, como a saída da função USR fornece o par de registros BC, você obterá o conteúdo do registro C mais 117*256 = 29952.

Vejamos agora outra variável do programa interpretador, chamada S-TOP (Screen TOP = topo da tela), que indica o número da linha de programa em BASIC que aparece no topo da tela quando o computador faz uma listagem automática (seu conteúdo está nos endereços 16419 e 16420 pois ela tem 2 bytes). A listagem automática é aquela obtida ao se pressionar NEW LINE sem o LIST. Assim, por exemplo, coloque o cursor na linha 8Ø do programa HEXAMEM:

LIST 8Ø

e então digite:

POKE 16419,5Ø (N.L.)
POKE 16420,Ø (N.L.)

e, a seguir, NEW LINE novamente; com isso, estamos fazendo S-TOP = 5Ø, ou seja, dizendo ao programa interpretador que o número da linha do topo da tela, em listagem automática, deve ser 5Ø. Observe o número da linha que está no topo da tela. Como esperado, ele deve ser

256*Ø + 5Ø = 5Ø

Digite agora:

LIST 545 (N.L.)

e, a seguir:

POKE 16419,244 (N.L.)
POKE 1642Ø,1 (N.L.)

e novamente NEW LINE. Qual a linha do topo da tela?

(1*256 + 244 = 5ØØ)

Veja que, para esse "truque" funcionar, é necessário que o cursor esteja *abaixo* da linha que desejamos ver no topo da tela.

Simularemos esses POKEs com uma sub-rotina em linguagem de máquina (digite *P* no final):

MEM.	3ØØØØ	LD A,244	'3EF4'	
MEM.	3ØØØ2	LD (16419),A	'32234Ø'	; (16419 = '4Ø23') coloca na memória 16419 o conteúdo de A, que é 244
MEM.	3ØØØ5	LD A,1	'3EØ1'	
MEM.	3ØØØ7	LD (1642Ø),A	'32244Ø'	; coloca na memória 1642Ø o conteúdo de A, que é 1
MEM.	3ØØ1Ø	RET	'C9'	

Prog. 2.10 – A variável S-TOP (endereços 16419 e 1642Ø)

Alexandre Fernandes
RG: 14.317.733

Coloque este programa na memória, faça LIST 545 e, a seguir, execute-o fazendo PRINT USR 3ØØØØ. Note que não importa o que essa sub-rotina "devolve" ao BASIC (no caso será 3ØØØØ, pois o par BC não foi alterado); sua função é simplesmente a de alterar o valor da variável S-TOP. De fato, pressione NEW LINE e veja o que ocorre! Novamente teremos a linha 5ØØ no topo da tela. Perceba que nos programas estamos colocando alguns comentários para facilitar a compreensão (sempre os colocaremos após um ponto e vírgula(;)).

Agora preste atenção ao código hexadecimal da instrução LD (16419),A. Seu primeiro byte deve ser '32', que é seu código de operação seguido do endereço invertido, ou seja, '234Ø'. Chamamos a atenção para o fato de que a tendência natural é escrever essa 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, '324A7C', essa seqüência será interpretada pelo microprocessador como LD ('7C4A'),A pois o número '32', quando interpretado como instrução, ordena que o microcomputador copie o conteúdo do acumulador na memória indicada pelos próximos 2 bytes (sempre levando em conta a inversão).

Ao escrever '4A7C32', o microprocessador interpretará o seguinte:

'4A' LD C,D
'7C' LD A,H
'32' LD (????),A

onde o endereço (????) 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 '4A', se interpretado como instrução, significa LD C,D; mas, se quisermos fazer LD ('7C4A'),A, o byte '4A' será parte dos dados da 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,(16388)? Como é possível copiar o conteúdo da memória 16388, que tem 1 byte apenas, no par de registros HL, que tem 2 bytes? De fato, essa instrução funciona da seguinte maneira: copia no registro L (o menos significativo) o conteúdo da memória 16388, e no registro H (o mais significativo) o conteúdo da memória seguinte, 16389. Isso em BASIC poderia ser associado a

LET X = PEEK 16388 + 256 * PEEK 16389

(você se lembra qual a variável contida nessas memórias?).

Note bem a diferença entre as seguintes instruções:

LD HL,16388
LD HL,(16388)

A segunda foi a que acabamos de descrever; a primeira foi descrita anteriormente e significa: carregue o par HL com o número 16388, ou seja, H = '40' e L = '04'.

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

LD (16388),HL

que copia o registro L na memória 16388 e o H na memória 16389 e que, em BASIC, equivaleria a:

POKE 16388,X – INT(X/256) * 256
POKE 16389,INT(X/256)

você saberia explicar por quê?

Assim temos as seguintes instruções:

INSTRUÇÃO	CÓDIGO
LD BC,(endereço)	'ED4B' + (2 bytes para o endereço)
LD DE,(endereço)	'ED5B' + (2 bytes para o endereço)
LD HL,(endereço)	'2A' + (2 bytes para o endereço)
LD (endereço),BC	'ED43' + (2 bytes para o endereço)
LD (endereço),DE	'ED53' + (2 bytes para o endereço)
LD (endereço),HL	'22' + (2 bytes para o endereço)

Tab. 2.6 – Endereçamento direto para pares de registros.

Note que aqui aparecem algumas instruções cujo código de operação tem 2 bytes. Para todas elas, o primeiro byte corresponde a 'ED'. 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 'ED' e outras, ainda, precedido por 'CB'. Por exemplo, o código '4B', sozinho, corresponde à instrução (LD C,E) e, precedido por 'ED', ou seja, 'ED4B', corresponde a LD BC,(endereço).

Observação: Não existe endereçamento indireto para pares de registros.

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

LD (HL),número ('36' + 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 da memória 30500 na memória 30510?

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

```
MEM. 30000  LD A,(30500)   '3A2477'  ; (30500 = '7724')
MEM. 30003  LD (30510),A   '322E77'  ; (30510 = '772E')
MEM. 30006  RET            'C9'
```

Prog. 2.11 – Programa para copiar um registro de memória em outro.

Teste o programa fazendo, por exemplo, POKE 30500,31 e, após executar esse programa, fazendo PRINT PEEK 30510, onde você deverá obter o número 31.

Exemplos de Aplicação

a) Utilizaremos a instrução LD (HL),dado:

```
MEM. 30000  LD HL,30009   '213975'  ; carrega o par HL com
                                      30009
MEM. 30003  LD (HL),'FF'  '36FF'    ; coloca 'FF' no endereço
                                      indicado por HL (30009)
```

MEM. 30005	LD C,(HL)	'4E'	; copia em C o conteúdo da memória cujo endereço é indicado por HL
MEM. 30006	LD B,0	'0600'	; coloca 0 em B para preparar a saída
MEM. 30008	RET	'C9'	

Prog. 2.12 – Uso do endereçamento indireto para DADOS.

Ao fazer PRINT USR 30000, XL ou XF (no programa HEXAMEM), deveremos obter na tela o número 255('FF'). Por que utilizamos a memória 30009 para colocar o número?

b) 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:

MEM. 30000	LD DE,(30010)	'ED5B3A75'	;	copia em E o conteúdo da memória 30010 e em D o da memória 30011
MEM. 30004	LD D,0	'1600'	;	"zera" o registro D
MEM. 30006	LD B,0	'0600'	;	"zera" o registro B
MEM. 30008	LD C,E	'4B'	;	copia o registro E em C
MEM. 30009	RET	'C9'		

Prog. 2.13 – Simulação da instrução LD E,(endereço).

Faça POKE 30010,100 e você deverá obter 100 na tela ao executar o programa.

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. Por exemplo, o último programa tinha 10 bytes e, portanto, começava na memória 30000 e terminava na memória 30009; assim, colocamos nossa variável na memória 30010.

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".

Por exemplo, se no programa anterior você fizesse POKE 30009,100, estaria substituindo a instrução RET código 'C9'(201) pelo número 100 e o computador, não encontrando a instrução RET, não poderia mais voltar ao programa em BASIC.

Resumo

Neste capítulo aprendemos o que é registro, par de registros e quais são os registros internos mais importantes do microprocessador: A, B, C, D, E, H, L. Podemos notar que esses registros permitem maior mobilidade que os da memória; por exemplo, existem instruções para copiar dados de um registro interno em outro mas *não* existem instruções para copiar diretamente dados de um lugar da memória em outro. Portanto, é obrigatório que o dado passe primeiro pelos registros internos do microprocessador. Vimos que existe um registro privilegiado chamado acumulador, que será estudado em detalhes adiante.

Observação: Perceba que é usual chamarmos apenas de *registros* os registros internos do microprocessador e de *memórias* os registros das memórias.

Aprendemos a instrução LD, que serve para colocar números diretamente nos registros ou na memória, para copiar dados de um registro em outro e para copiar dados da memória em um registro, e vice-versa, usando endereçamento direto ou indireto.

Observação: Lembre-se que as instruções LD registro,registro ou LD memória,registro *não* alteram o valor do registro que está sendo copiado, pois seu conteúdo é apenas copiado no lugar de destino.

Os programas em linguagem de máquina devem terminar sempre com a instrução RET (e nunca HALT) e ser tratados como sub-rotinas de um programa em BASIC, chamado através da função USR que fornece como saída o conteúdo do par de registro BC, ou seja, 256 * (conteúdo de B) + (conteúdo de C).

Vimos também quatro novas variáveis do programa interpretador do TK (ROM): a que indica a linha de PRINT (1 a 24) no endereço 16442, a que indica a coluna (endereço 16441), a que indica S-TOP nos endereços 16419 e 16420 (SCREEN TOP, em listagem automática) e a que indica a linha para a qual CONT "salta" (endereços 16427 e 16428), que será utilizada nos exercícios.

Exercícios

1. Faça um programa em linguagem de máquina que copie o conteúdo da memória 30015 na memória 30016. Primeiramente, coloque o número 30015 no par HL e o número 30016 no par DE. Feito isto, copie o conteúdo da memória indicada por HL no acumulador e o conteúdo do acumulador na memória indicada por DE. A seguir, copie o acumulador no registro C e "zere" o registro B. Assim, ao executar o programa você irá obter na tela o número que estava na memória 30015.
   Obviamente, antes de executar coloque um número na memória 30015, por exemplo:

   POKE 30015,99

   e, uma vez executado o programa, verifique se ele funcionou fazendo:

   PRINT PEEK 30016

   e você deverá obter 99.

Observação: É sempre bom lembrar que colocamos as variáveis em endereços que ficam *após* o fim do programa, ou seja, após RET('C9'). Existem meios mais simples do que aqueles sugeridos para fazer este programa; no entanto, achamos essa maneira mais didática por estarmos ainda no segundo capítulo.

2. Responda às seguintes perguntas:
   a) Por que nas instruções de LD nunca aparecem registros simples entre parênteses, apenas pares de registros?
   b) Se o par HL contém o valor 16434, qual a diferença entre as instruções LD B,(HL), LD BC,(16434) e LD BC,16434?
   c) Tente fazer um programa em linguagem de máquina que copie no par de registros HL o valor da memória 16442 (note que H deverá ser zero) usando, para lidar com a memória, apenas instruções do tipo LD registro,(HL).
   d) Suponha que o acumulador contenha o número 24 e que o par HL também contenha 24 (H = '00' e L = '18'). Procure analisar o efeito das seguintes instruções:

LD A,24	LD A,(24)
LD H,24	LD A,(HL)
LD L,24	LD (24),A
LD HL,24	LD (HL),A
LD L,A	LD HL,(24)
LD A,H	LD (24),HL
	LD (HL),24

   e) *Se existisse*, qual seria o efeito da instrução LD BC,(HL)?

3. Procure fazer um programa em BASIC cheio de PRINTs com pausas (PAUSE) entre os mesmos. Interrompa o programa usando BREAK e, a seguir, altere o valor das memórias 16427 e 16428. Nessas memórias fica guardada a linha para a qual o programa deve ir após ser pressionado o CONT.
   Faça as alterações usando POKEs diretamente e, a seguir, usando programa em linguagem de máquina (lembre-se da inversão para os números maiores que 255). Verifique se após o CONT o programa vai para a linha que você especificou. Por que esta variável deve ter 2 bytes? (Ver programa 2.10.)

Observação: Para ajudá-lo a fazer os programas, temos, no final deste livro, o apêndice 2, contendo todas as instruções (mnemônicos) e seus respectivos códigos hexadecimais. Obviamente, você não precisará decorar os códigos das instruções, apenas seus mnemônicos.

Instruções vistas neste capítulo

RET	LD registro,(HL)
HALT	LD (endereço),A
LD registro, dado de 8 bits	LD (BC),A
LD par de registros,dado de 16 bits (invertido)	LD (DE),A
LD registro,registro	LD (HL),registro
LD A,(endereço)	LD par de registros,(endereço)
LD A,(BC)	LD (endereço),par de registros
LD A,(DE)	LD (HL),dado de 8 bits

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.

ADIÇÃO: AS INSTRUÇÕES ADD, ADC E INC E A FLAG DE CARRY

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

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

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

Tab. 3.1 – A instrução ADD entre registros.

que significam: some o conteúdo do registro ou par de registros da direita com o da 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 tentar somar o conteúdo do registro D com o registro E, utilizando uma sub-rotina em linguagem de máquina:

```
MEM. 30000   LD E,34    '1E22'
MEM. 30002   LD D,66    '1642'
MEM. 30004   LD A,D     '7A'     ; copia D em A
MEM. 30005   ADD A,E    '83'     ; soma E com A, ou seja, com D
MEM. 30006   LD C,A     '4F'     ; copia o resultado em C
MEM. 30007   LD B,0     '0600'   ; coloca 0 em B para a saída do pro-
                                   grama
MEM. 30009   RET        'C9'
```

Prog. 3.1 – Programa para somar os registros D e E.

Note que a soma *não* pode ser feita diretamente. Devemos, obrigatoriamente, usar o acumulador. Coloque o programa na memória e execute-o (usando HEXAMEM). 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 'FF') que é o *máximo* que "cabe" num registro? E, no caso de pares de registros, se o resultado for maior que 65535 ('FFFF')?

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:

```
MEM. 30000   LD HL,1        '210100'   ; coloca 1 em HL, ou seja, '0001'
                                         (note que devemos completar
                                         os 2 bytes de dados)
MEM. 30003   LD BC,65535    '01FFFF'   ; coloca o valor máximo em BC,
                                         ou seja, 'FFFF'
MEM. 30006   ADD HL,BC      '09'       ; soma BC com HL; o resultado
                                         fica em HL
MEM. 30007   LD B,H         '44'       ; coloca H em B e L em C para
                                         poder ter o resultado na tela
MEM. 30008   LD C,L         '4D'
MEM. 30009   RET            'C9'
```

Prog. 3.2 – Soma de 65535 com 1.

Você obtém como resultado o número Ø. De fato, pensando em hexadecimal, ao somar '1' ao número 'FFFF' deveríamos obter '10000'; mas, veja 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, Ø.

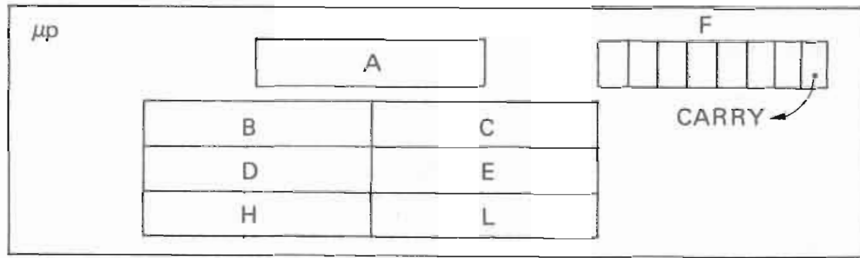


Fig. 3.1 – O registro interno F e a flag de CARRY.

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:

INSTRUÇÃO	CÓDIGO
ADC A,A	'8F'
ADC A,B	'88'
ADC A,C	'89'
ADC A,D	'8A'
ADC A,E	'8B'
ADC A,H	'8C'
ADC A,L	'8D'

INSTRUÇÃO	CÓDIGO
ADC HL,BC	'ED4A'
ADC HL,DE	'ED5A'
ADC HL,HL	'ED6A'

Observação: Estas três instruções fazem parte das instruções precedidas por 'ED', mencionadas no capítulo anterior.

Tab. 3.2 – A instrução ADC.

Vamos então refazer o programa 3.1 para executar uma soma cujo resultado seja maior que 255:

```
MEM. 30000   LD E,200    '1EC8'
MEM. 30002   LD D,58     '163A'
MEM. 30004   LD A,D      '7A'
MEM. 30005   ADD A,E     '83'     ; soma E com A e gera CARRY
MEM. 30006   LD C,A      '4F'     ; coloca o resultado em C
MEM. 30007   LD A,Ø      '3EØØ'   ; coloca Ø em A
MEM. 30009   ADC A,A     '8F'     ; transfere o CARRY para A e a seguir
                                    para B
MEM. 30010   LD B,A      '47'
MEM. 30011   RET         'C9'
```

Prog. 3.3 – Soma de dois registros com resultado maior que 255.

Note que ao somar 200 com 58 obteremos 258, ou seja, A = 2 e CARRY = 1. A seguir, este CARRY é transferido para o acumulador (ADC A,A) e colocado em B; desse modo, teremos B = 1 e C = 2, ou seja, BC = 256 * 1 + 2 = 258.

Para exemplificar o raciocínio a ser adotado, execute os seguintes programas, que somam 13189 com 31687:

a) Procedimento usando pares de registros:

```
MEM. 30000   LD DE,13189   '118533'   ; coloca 13189 em DE
MEM. 30003   LD HL,31687   '21C77B'   ; coloca 31687 em HL
MEM. 30006   ADD HL,DE     '19'       ; soma DE com HL
MEM. 30007   LD B,H        '44'       ; coloca H em B e L em C para
                                        a saída na tela
MEM. 30008   LD C,L        '4D'
MEM. 30009   RET           'C9'
```

Prog. 3.4 – Soma dos pares DE com HL.

b) Soma por partes usando ADC:

```
MEM. 30000   LD D,51     '1633'   ; note que: 256*51 + 133 = 13189
MEM. 30002   LD E,133    '1E85'   ; coloca 13189 em DE (equivalem a
                                    LD DE,13189)
MEM. 30004   LD H,123    '267B'   ; note que: 256* 123 + 199 = 31687
MEM. 30006   LD L,199    '2EC7'   ; coloca 31687 em HL (equivalem a
                                    LD HL,31687)
MEM. 30008   LD A,L      '7D'     ; coloca L em A
MEM. 30009   ADD A,E     '83'     ; soma E com A e gera CARRY
MEM. 30010   LD L,A      '6F'     ; coloca o resultado em L
MEM. 30011   LD A,H      '7C'     ; coloca H em A
MEM. 30012   ADC A,D     '8A'     ; soma D com A e com CARRY
MEM. 30013   LD H,A      '67'     ; coloca o resultado em H
MEM. 30014   LD B,H      '44'     ; transfere o resultado para BC para
                                    saída na tela
MEM. 30015   LD C,L      '4D'
MEM. 30016   RET         'C9'
```

Prog. 3.5 – Soma de 13189 com 31687, por partes, pois os números têm 2 bytes cada.

Como esperado, os dois programas produzem o mesmo resultado, ou seja, 44876 (= 13189 + 1687). Duas coisas devem ser observadas: inicialmente, que as instruções LD *não* afetam o valor do CARRY pois, caso contrário, no segundo programa as duas 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 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 30012,130 (130 = '82', que é o código de ADD A,D)

Execute o programa PRINT USR 30000. Você obterá 44620, que é um resultado incorreto. Veja porque: os conteúdos dos registros eram, respectivamente:

D = '33'	H = '7B'
E = '85'	L = 'C7'
('3385' = 13189)	('7BC7' = 31687)

Observação: Tente fazer as somas que se seguem em hexadecimal; não se esqueça que "vai um" só ocorre quando os dígitos somados resultarem maiores que 'F'

(= 15) e que para achar o que "fica" devemos subtrair '10' (= 16) do resultado. Caso você não consiga, transforme os números em decimal, faça a soma e a seguir converta o resultado hexadecimal. Exemplo:

```
     11
    '4A'
+   'C8'
  ------
   '112'
```

Assim, ao efetuar

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

obtemos:

L = 'C7' + '85' = '4C' (CARRY = 1)

e ao efetuar

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

obtemos:

H = '7B' + '33' + '1' = 'AE' + '1' = 'AF'

E o resultado seria: H = 'AF4C'

ou seja: 256*175 + 76 = 44876

Se fizermos ADD A,D, teremos H = 'AE' (ou H = 174); ou seja, HL = 'AE4C', 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 utilizado no segundo programa, ou seja, somar por partes. Vamos supor que quiséssemos somar os seguintes números:

'02A5EF' (= 2*65.536 + 165*256 + 239 = 173.551)

com 'F4DE' (= 244*256 + 222 = 62.686)

(*Observação:* $65.536 = (256)^2$.)

deveremos 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 nas memórias após o fim do programa.

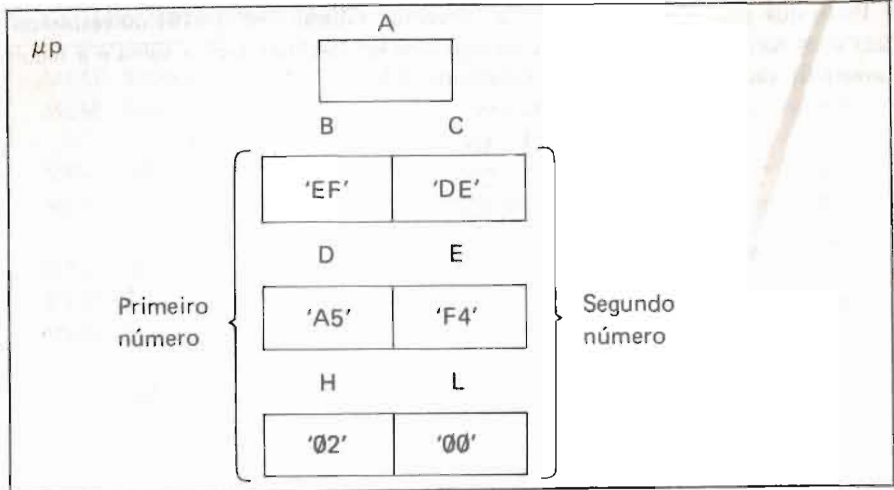
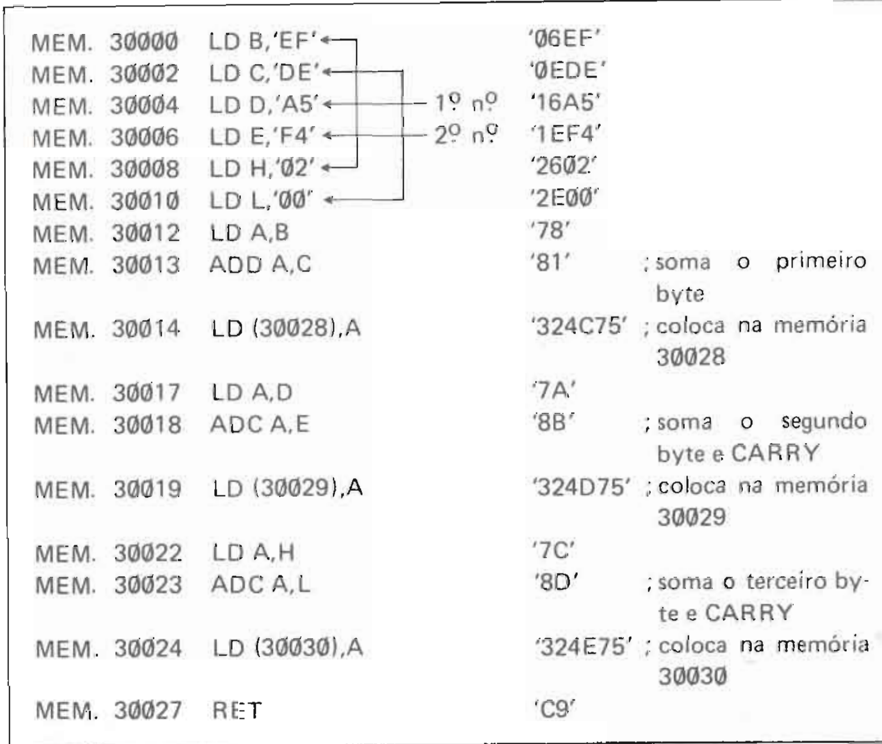


Fig. 3.2 – Visualização do microprocessador.



```
MEM. 3ØØØØ   LD B,'EF'                 'Ø6EF'
MEM. 3ØØØ2   LD C,'DE'                 'ØEDE'
MEM. 3ØØØ4   LD D,'A5'     — 1º nº     '16A5'
MEM. 3ØØØ6   LD E,'F4'     — 2º nº     '1EF4'
MEM. 3ØØØ8   LD H,'Ø2'                 '26Ø2'
MEM. 3ØØ1Ø   LD L,'ØØ'                 '2EØØ'
MEM. 3ØØ12   LD A,B                    '78'
MEM. 3ØØ13   ADD A,C                   '81'      ; soma o primeiro
                                                   byte
MEM. 3ØØ14   LD (3ØØ28),A              '324C75'  ; coloca na memória
                                                   3ØØ28
MEM. 3ØØ17   LD A,D                    '7A'
MEM. 3ØØ18   ADC A,E                   '8B'      ; soma o segundo
                                                   byte e CARRY
MEM. 3ØØ19   LD (3ØØ29),A              '324D75'  ; coloca na memória
                                                   3ØØ29
MEM. 3ØØ22   LD A,H                    '7C'
MEM. 3ØØ23   ADC A,L                   '8D'      ; soma o terceiro by-
                                                   te e CARRY
MEM. 3ØØ24   LD (3ØØ3Ø),A              '324E75'  ; coloca na memória
                                                   3ØØ3Ø
MEM. 3ØØ27   RET                       'C9'
```

Prog. 3.6 – Soma por partes de dois números com 3 bytes cada.

Novamente não nos interessa o resultado final dos registros BC, que no caso é 614Ø6. O resultado da nossa soma por partes está nas memórias 3ØØ28, 3ØØ29 e 3ØØ3Ø, que estão *após* a instrução de RET. Portanto, para verificar se o nosso raciocínio está certo, basta fazer

PRINT 65536 * PEEK 3ØØ3Ø + 256 * PEEK 3ØØ29 + PEEK 3ØØ28

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 as memórias 3ØØ28, 3ØØ29 e 3ØØ3Ø para colocar os resultados por estarem *após* o programa.

Fig. 3.3 – Visualização da memória.

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

INSTRUÇÃO	CÓDIGO
ADD A,dado	'C6' + 1 byte de dados
ADC A,dado	'CE' + 1 byte de dados

Tab. 3.3 – Soma direta de números ao acumulador.

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, 60000 em HL). Isso poderia ser feito da seguinte maneira:

```
MEM. 30000  LD HL,60000   '2160EA'  ; carrega HL com 60000
MEM. 30003  LD DE,74      '114A00'  ; carrega DE com número a ser
                                      somado (74, no caso)
MEM. 30005  ADD HL,DE     '19'      ; soma DE com HL
MEM. 30006  LD, B,H       '44'      ; transfere o resultado para BC
MEM. 30007  LD C,L        '4D'
MEM. 30008  RET           'C9'
```

Prog. 3.7 – Soma do número 74 com o par HL (60000).

Deveremos obter 60074. Observe que tivemos de utilizar o par de registros DE, colocando '00' 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:

```
MEM. 30000  LD   HL,60000  '2160EA'  ; carrega HL com 60000
MEM. 30003  LD   A,L       '7D'      ; coloca L no acumulador
MEM. 30004  ADD  A,74      'C64A'    ; soma diretamente 74 e gera
                                       CARRY
MEM. 30006  LD   L,A       '6F'      ; coloca o resultado em L
MEM. 30007  LD   A,H       '7C'      ; coloca H em A
MEM. 30008  ADC  A,0       'CE00'    ; soma o CARRY ("vai um")
MEM. 30010  LD   H,A       '67'      ; coloca o resultado em H
MEM. 30011  LD   B,H       '44'      ; transfere o resultado para BC
MEM. 30012  LD   C,L       '4D'
MEM. 30013  RET            'C9'
```

Prog. 3.8 – Soma do número 74 com HL (60000) sem usar outros pares.

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 adição 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:

INSTRUÇÃO	CÓDIGO
INC A	'3C'
INC B	'04'
INC C	'0C'
INC D	'14'
INC E	'1C'
INC H	'24'
INC L	'2C'
INC BC	'03'
INC DE	'13'
INC HL	'23'

Tab. 3.4 – A instrução INC para os registros internos.

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:

INSTRUÇÃO	CÓDIGO
ADD A,(HL)	'86'
ADC A,(HL)	'8E'
INC (HL)	'34'

Tab. 3.5 – As instruções INC e ADD usando a memória.

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 da memória 30010 (por exemplo, 36):

```
MEM. 30000   LD    HL,30010  '213A75'  ; coloca 30010 em HL
MEM. 30003   LD    A,64      '3E40'    ; coloca 64 em A
MEM. 30005   ADD   A,(HL)    '8E'      ; soma o conteúdo da memória 30010 com A
MEM. 30006   LD    B,0       '0600'    ; transfere o resultado para BC
MEM. 30008   LD    C,A       '4F'
MEM. 30009   RET             'C9'
```

Prog. 3.9 – Soma do número 64 à memória 30010.

Faça POKE 30010,36 e, a seguir, 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 de RET (30010, no nosso caso). Entretanto, para programas mais complexos, é interessante colocar as variáveis *bem depois* da instrução de 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?

SUBTRAÇÃO: AS INSTRUÇÕES SUB, SBC E DEC

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

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

'97' → Corresponde a fazer o acumulador igual a *zero*.

Tab. 3.6 – A instrução SUB (SUBtract) para os registros.

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 intruções:

INSTRUÇÃO	CÓDIGO
SBC A,A	'9F'
SBC A,B	'98'
SBC A,C	'99'
SBC A,D	'9A'
SBC A,E	'9B'
SBC A,H	'9C'
SBC A,L	'9D'
SBC HL,BC	'ED42'
SBC HL,DE	'ED52'
SBC HL,HL	'ED62'

Tab. 3.7 – A instrução SBC para registros.

Alexandre Fernandes
RG: 14.317.733

Façamos uma sub-rotina para "zerar" o par de registros HL:

```
MEM. 30000   ADD   A,0     'C600'   ; gera CARRY = 0
MEM. 30002   SBC   HL,HL   'ED62'   ; subtrai HL de HL e subtrai CARRY (= 0)
MEM. 30004   LD    B,H     '44'     ; transfere o resultado para BC
MEM. 30005   LD    C,L     '4D'
MEM. 30006   RET           'C9'
```

Prog. 3.10 – Programa para "zerar" HL sem utilizar LD.

Note que a instrução ADD A,0 foi utilizada para garantir que a flag de CARRY fosse para 0 (zero). Tente imaginar o que aconteceria se não colocássemos esta instrução e o CARRY estivesse com o valor 1.

Podemos também subtrair constantes numéricas do acumulador:

INSTRUÇÃO	CÓDIGO
SUB A,dado	'D6' + 1 byte para o dado
SBC A,dado	'DE' + 1 byte para o dado

Tab. 3.8 – Subtração de números do acumulador.

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
('02A5EF') – ('F4DE')

```
MEM. 30000  LD   B,'EF'      ; primeiro número em H, D e B
MEM. 30002  LD   C,'DE'      ; segundo número em L, E e C
MEM. 30004  LD   D,'A5'
MEM. 30006  LD   E,'F4'
MEM. 30008  LD   H,'02'
MEM. 30010  LD   L,'00'
MEM. 30012  LD   A,B
MEM. 30013  SUB  A,C         ; subtrai o primeiro byte
MEM. 30014  LD   (30028),A   ; coloca o resultado na memória 30028
MEM. 30017  LD   A,D
MEM. 30018  SBC  A,C         ; subtrai com CARRY o segundo byte
MEM. 30019  LD   (30029),A   ; coloca o resultado na memória 30029
MEM. 30022  LD   A,H
MEM. 30023  SBC  A,L         ; subtrai com CARRY o terceiro byte
MEM. 30024  LD   (30030),A   ; coloca o resultado na memória 30030
MEM. 30027  RET
```

Prog. 3.11 – Subtração por partes de dois números de 3 bytes cada.

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 nos 3 bytes de memória, 30028, 30029 e 30030, corresponde ao resultado esperado.

Vale a pena lembrar que não existe instrução para subtrair diretamente um registro de um par ou 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 60000, e 266 de HL carregado com 60000.) Finalmente, para terminar nossa analogia com a adição, introduziremos as instruções de DEC (abreviação de DECrement), e a subtração utilizando as memórias:

INSTRUÇÃO	CÓDIGO
DEC A	'3D'
DEC B	'05'
DEC C	'0D'
DEC D	'15'
DEC E	'1D'
DEC H	'25'
DEC L	'2D'
DEC BC	'0B'
DEC DE	'1B'
DEC HL	'2B'
DEC (HL)	'35'
SUB A,(HL)	'96'
SBC A,(HL)	'9E'

Tab. 3.9 – A instrução DEC e a subtração com as memórias. As instruções de DEC, analogamente às de INC, *não* afetam o CARRY.

A instrução DEC subtrai 1 (um) do valor de dado registro (ou par) ou memória, 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 a flag de CARRY, vamos supor que queremos subtrair o conteúdo do par DE do par HL, sem alterar o acumulador (ou os registros B e C). Compare os seguintes programas:

```
INC   A       '3C'
DEC   A       '3D'
SBC   HL,DE   'ED52'
```

Prog. 3.12 – Simulação errada de um SUB para pares de registros com SBC.

e

```
ADD   A,0     'C600'
SBC   HL,DE   'ED52'
```

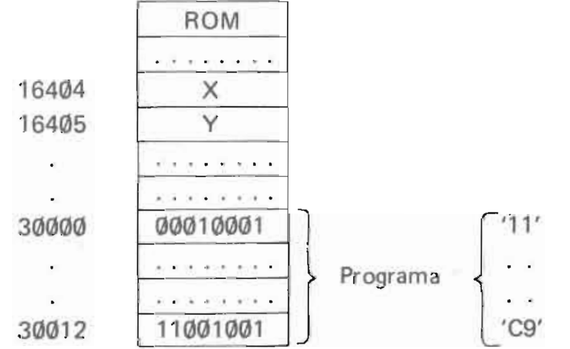
Prog. 3.13 – Simulação correta de um SUB para pares de registros com SBC.

Eles têm o mesmo 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 Ø (zero) devido à instrução ADD A,Ø). Lembre-se que as instruções INC e DEC *não* afetam o CARRY...

Iremos agora falar sobre mais uma variável do programa interpretador: E-LINE, variável de 2 bytes localizada nos endereços 164Ø4 e 164Ø5. Essa variável indica quantos bytes de memória ocupam o programa BASIC (que eventualmente você colocou no computador), as variáveis do mesmo e a tela de TV + 165Ø9; assim, basta fazer PEEK 164Ø4 + 256 * PEEK 164Ø5 e subtrair 165Ø9 para saber o tamanho do programa BASIC que está na memória (o porquê desta subtração será explicado no capítulo 8, quando daremos mais detalhes sobre a memória RAM do TK). Podemos fazer isso com a seguinte sub-rotina:

```
MEM. 3ØØØØ  LD   DE,165Ø9     '117D4Ø'  ; coloca 165Ø9 em DE
MEM. 3ØØØ3  LD   HL,(164Ø4)   '2A144Ø'  ; coloca em L o conteúdo
                                          da memória 164Ø4 e em
                                          H o conteúdo da memó-
                                          ria 164Ø5
MEM. 3ØØØ6  ADD  A,Ø          'C6ØØ'    ; "zera" CARRY (veja
                                          exemplos anteriores)
MEM. 3ØØØ8  SBC  HL,DE        'ED52'    ; subtrai DE de HL
MEM. 3ØØ1Ø  LD   B,H          '44'      ; transfere o resultado para
                                          BC
MEM. 3ØØ11  LD   C,L          '4D'
MEM. 3ØØ12  RET               'C9'
```

Prog. 3.14 – Cálculo do tamanho de um programa.



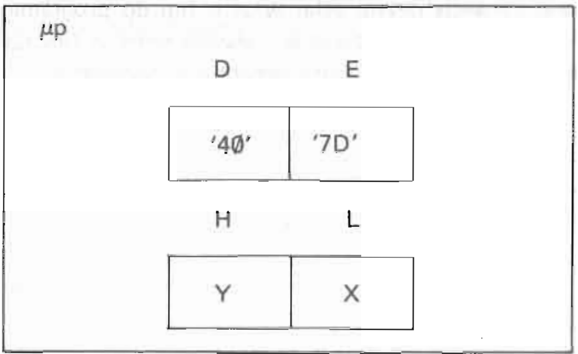


Fig. 3.4 – Visualização da memória e do microprocessador até a instrução ADD A,Ø.

Lembre-se que a instrução LD HL,(164Ø4) funciona da seguinte maneira: coloca o conteúdo da memória 164Ø4 no registro L e o conteúdo da memória 165Ø5 no registro H. Você deve estar, neste momento, com o programa HEXAMEM no computador; utilize essa sub-rotina para saber quantos bytes ele ocupa.

Resumo

Aprendemos a fazer adições e subtrações (embora não tenhamos ainda "entrado" nos números negativos), verificando que o acumulador é o registro privilegiado (em se tratando de pares, seria o par HL) por ser o alvo obrigatório das operações aritméticas (e dos endereçamentos diretos e indiretos via BC e DE). (Rever capítulo 2.)

Vimos que o CARRY (bit pertencente ao registro F do microprocessador) serve para indicar "vai um" (soma) ou "menor que zero" (subtração) e permite fazer adições ou subtrações por partes para números com mais de 2 bytes. O CARRY é gerado pelas instruções ADD registro, registro; ADC registro,registro; SUB registro,registro; e, SBC registro,registro.

Vimos também as instruções INC e DEC, notando que, assim como as instruções de LD, elas *não* afetam a flag de CARRY, deixando-a em seu valor anterior.

Quanto às variáveis do programa interpretador, verificamos que as memórias 164Ø4 e 164Ø5 (E-LINE) contêm o número de bytes (+ 165Ø9) ocupados pelo(s) programa(s) em BASIC, mais as variáveis correspondentes e a tela de TV.

Foi dada ênfase à estrutura dos programas: endereços das memórias à esquerda, seguidos da instrução, do código e do comentário precedido por ponto e vírgula. Os endereços são estritamente necessários quando se usa variáveis

na memória; essas variáveis devem estar *após* o fim do programa e, para programas complexos, é conveniente deixar um espaço entre o fim do programa e as variáveis, para eventuais modificações no tamanho do programa.

Exercícios

(Utilize o apêndice 2 para auxiliá-lo.)

1. Execute o seguinte programa:

```
LD  BC,Ø
LD  HL,Ø
ADD HL,BC
LD  B,H
LD  C,L
RET
```

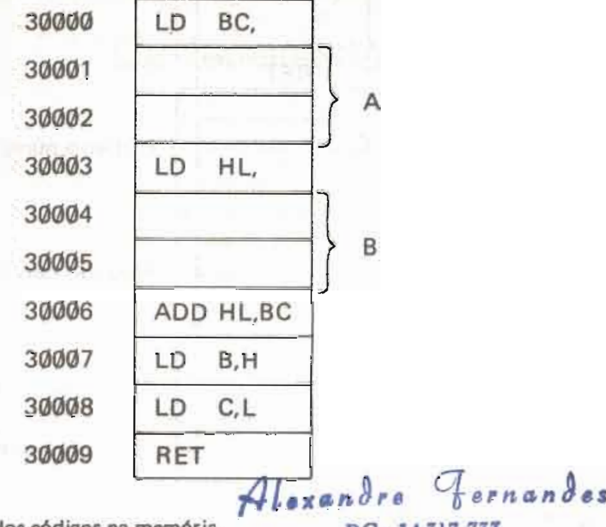
Coloque você mesmo os códigos em linguagem de máquina. O que você obteve? Se obteve zero, tudo bem; se obteve 13856, você fez um erro fundamental. Repare que as instruções LD BC,Ø e LD HL,Ø necessitam ter, obrigatoriamente, 3 bytes. Tente refazer o programa, ou melhor, reescrever os códigos. Não se esqueça de colocar os números das memórias à esquerda das instruções.

2. Execute agora, usando a sub-rotina do exercício 1, o seguinte programa:

```
5000 INPUT A
5005 INPUT B
5010 POKE 30001,A-INT (A/256)*25
6
5015 POKE 30002,INT (A/256)
5020 POKE 30004,B-INT (B/256)*25
6
5025 POKE 30005,INT (B/256)
5028 SCROLL
5030 PRINT A;"+";B;"=";
5035 PRINT USR 30000
5038 SCROLL
5040 PRINT
5045 GOTO 5000
```

Este programa colocará a variável A no segundo e terceiro bytes da sub-rotina em linguagem de máquina do exercício 1 e a variável B no quinto e sexto, ou seja, você estará dando valores aos pares BC e HL, que serão somados. Experimente usar valores maiores que 65535 (faça, por exemplo, 65535 + 1 = ?).

Note que esse programa exemplifica como passar parâmetros através do BASIC para uma sub-rotina em linguagem de máquina.



Alexandre Fernandes
RG: 14.317.733

Fig. 3.5 – Visualização dos códigos na memória.

Agora tente escrever um programa em BASIC que use uma outra sub-rotina em linguagem de máquina e que imprima uma tabela de valores de A e B e o resultado da sua subtração; tente subtrair números para obter resultados negativos... (Faça, por exemplo, Ø – 1 = ?.)

3. Escreva um programa em linguagem de máquina que produza como saída o número 1, se o par BC for maior ou igual a DE, e o número Ø, caso contrário.

4. Faça um programa em linguagem de máquina que deixe a flag de CARRY em 1 *sem* afetar nenhum registro! (ou seja, deixando seus conteúdos iniciais iguais aos finais).

   Observação: Faça isto sem utilizar a instrução ADD A,Ø.

5. Faça um programa em linguagem de máquina que some (ou subtraia) dois números de 4 bytes cada colocados nos 8 bytes *após* o fim do programa. O resultado deverá ser colocado nos 4 bytes seguintes aos dados (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 endereçamento por HL para colocar o resultado na memória, utilizando, portanto, INC HL.

Fig. 3.6 – Visualização da memória.

Instruções vistas neste capítulo

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

(A notação registro *não* inclui o registro F.)

Deslocamento de Blocos

AS INSTRUÇÕES LDI, LDIR, LDD e LDDR

Deslocar blocos significa deslocar de uma só vez os conteúdos de uma grande quantidade de memórias. Suponha que você tivesse uma sub-rotina razoavelmente grande (por exemplo, de 200 bytes), colocada a partir da memória 30000, e você quisesse transferi-la para começar na memória 20000. Se você tivesse *bastante paciência*, poderia fazer o seguinte:

```
LD    DE,20000    ; carrega DE com 20000 para servir de ponteiro de
                    destino
LD    HL,30000    ; carrega HL com 30000 para servir de ponteiro de
                    fonte
LD    A,(HL)      ; transfere de fonte para destino
LD    (DE),A
INC   HL          ; incrementa ponteiros
INC   DE
LD    A,(HL)      ; transfere de fonte para destino
LD    (DE),A
INC   HL          ; incrementa ponteiros
INC   DE
. . .   e assim por diante 200 vezes.
```

Prog. 4.1 – Transferência de 200 bytes de memória.

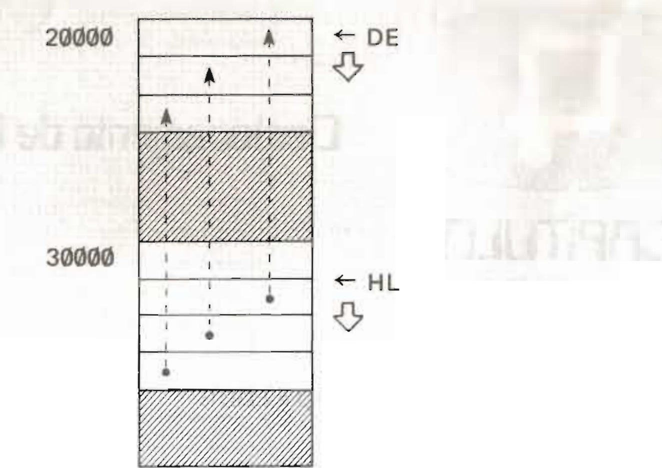


Fig. 4.1 – Visualização da memória na transferência.

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 memórias endereçadas por HL a DE. Por exemplo, a transferência das memórias endereçadas por HL para as endereçadas por BC não pode ser feita numa única instrução.

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

```
LD   DE,2ØØØØ   '11204E'   ; carrega ponteiros de memória
LD   HL,3ØØØØ   '213075'
LDI             'EDAØ'     ; faz transferência e incrementa ponteiros
LDI             'EDAØ'
. . . e assim por diante 2ØØ vezes.
```

Prog. 4.2 – Transferência usando 2ØØ vezes LDI.

Isto reduz bastante o programa, mas há ainda uma maneira melhor de se 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 quiséssemos mover 2ØØ bytes, deveríamos fazer:

```
LD   BC,2ØØ     'Ø1C8ØØ'   ; carrega contador
LD   DE,2ØØØØ   '11204E'   ; carrega ponteiros de memória
LD   HL,3ØØØØ   '213Ø75'
LDIR            'EDBØ'     ; transfere, incrementa ponteiros, de-
                             crementa o contador e checa o final
                             do loop (BC = Ø)
```

Prog. 4.3 – Transferência usando LDIR.

Existem também as instruções LDD (código 'EDA8') e LDDR (código 'EDB8') 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.

APLICAÇÃO PRÁTICA: SCROLL E ANTISCROLL

Apresentaremos um programa que nos permitirá fazer um SCROLL ao contrário. Em outras palavras, vamos fazer a tela *descer* em vez de *subir*. Entretanto, para que ele funcione, é necessária uma expansão de memória...

Apenas uma explicação prévia quanto à tela: como todos sabemos, podemos imprimir até 22 linhas com 32 caracteres cada uma. Entretanto, no fim de cada linha o computador introduz um código de NEW LINE; portanto, *na memória*, a tela ocupa 22 x 33 = 726 bytes (que são armazenados *seqüencialmente*). Estes NEW LINE's são invisíveis na tela e servem para indicar ao TK o fim de cada linha.

Observação: Não estamos considerando as duas linhas de edição.

Observe o seguinte programa (coloque-o a partir da memória 3ØØØØ):

MEM. 30000	LD BC,726	'01D602'	; carrega BC com o "tamanho" da tela
MEM. 30003	LD HL,(16396)	'2A0C40'	; carrega em HL o valor de D-FILE
MEM. 30006	ADD HL,BC	'09'	; soma HL com BC para obter o endereço do fim da tela
MEM. 30007	LD D,H	'54'	; transfere para DE o endereço do fim da tela
MEM. 30008	LD E,L	'5D'	
MEM. 30009	LD BC,693	'01B502'	; carrega BC com o "tamanho" de 21 linhas (21 x 23)
MEM. 30012	LD HL,(16396)	'2A0C40'	; carrega em HL o valor de D-FILE
MEM. 30015	ADD HL,BC	'09'	; soma HL com BC para obter o endereço do fim da penúltima linha
MEM. 30016	LDDR	'EDB8'	; transfere HL para DE, decrementa ponteiros e contador e testa fim de loop (BC = 0)
MEM. 30018	RET	'C9'	

Prog. 4.4 — ANTISCROLL.

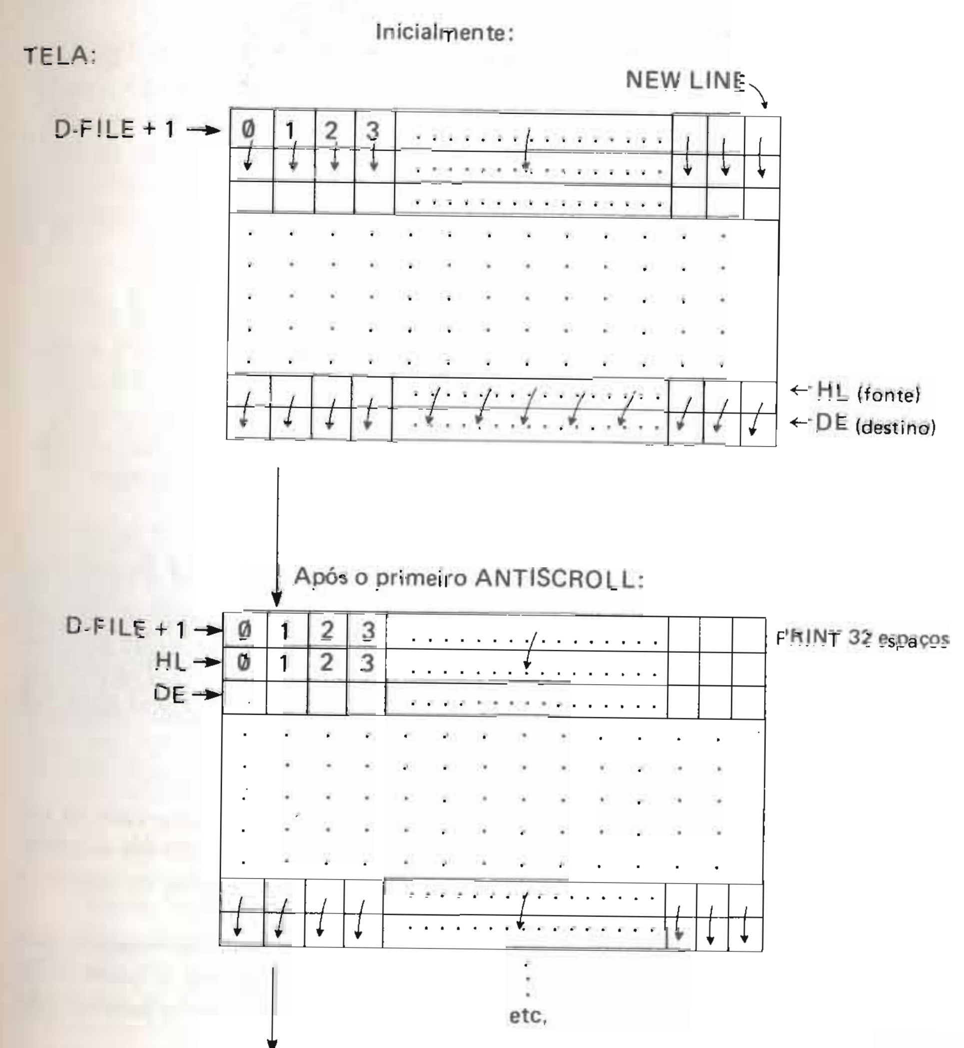
Não execute o programa ainda (digite P). Leia antes a explicação que se segue.

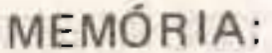
A primeira instrução carrega 726 em BC, que é o número de posições na memória da tela (contando os espaços vazios e os NEW LINEs). Nos endereços da RAM 16396 e 16397 está outra variável do programa interpretador, chamada D-FILE, que contém o endereço inicial do conteúdo da tela na memória RAM, *menos um*. Dessa forma, ao carregarmos HL com o conteúdo de D-FILE, quando somamos 726, calcularemos o endereço do último caractere da tela; este endereço é então colocado no par DE (lembre-se de que necessitamos de duas instruções para carregar o par HL no par DE, pois *não* existe a instrução LD DE,HL).

Colocaremos agora em HL o endereço do caractere que *será* o último da tela após fazermos este ANTISCROLL, ou seja, o endereço do último caractere da penúltima linha (pois a última será perdida), que seria D-FILE + 21 * 33 = = D-FILE + 693, o que é feito carregando novamente HL com D-FILE e somando

693. Note que agora o "terreno" está preparado para a instrução LDDR, pois temos em BC precisamente o número de caracteres que deverão ser deslocados (693), em HL (fonte) o endereço do último caractere da 21ª linha e em DE (destino) o endereço do último caractere da 22ª linha; basta apenas "apagar" a primeira linha da tela, o que pode ser feito utilizando um PRINT AT 0,0; "32 espaços".

A figura 4.2 mostra os ponteiros HL, DE e D-FILE na tela de TV antes e depois da execução da sub-rotina e o andamento de HL e DE na memória.





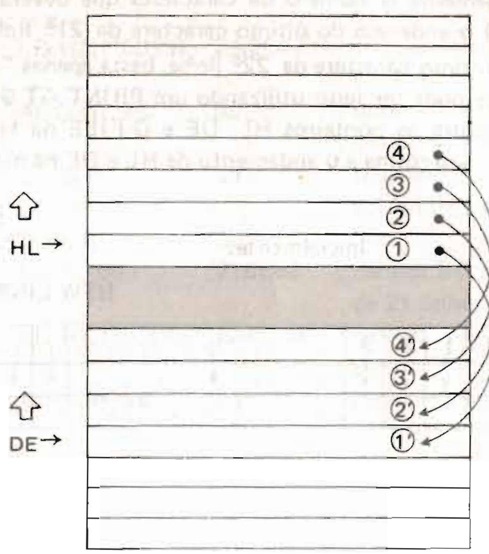


Fig. 4.2 – Visualização da transferência da memória do ANTISCROLL.

Execute então o seguinte programa:

```
3000 SLOW
3010 FOR I=1 TO 100
3020 PRINT AT USR 30000,0;"                                "
3030 PRINT I
3040 NEXT I
```

Prog. 4.5 – Demonstração de ANTISCROLL.

Que tal o efeito? Preste atenção à instrução 3Ø2Ø: qual será o valor de BC no final da sub-rotina em linguagem de máquina? Será Ø, pois BC era o contador! Assim, PRINT AT USR 3ØØØØ,Ø equivale a PRINT Ø,Ø, além de chamar a sub-rotina que está a partir da memória 3ØØØØ.

Vamos agora fazer uma outra sub-rotina, em linguagem de máquina, que produz o efeito de SCROLL normal. Para isto, devemos carregar o ponteiro de fonte (HL) com D-FILE + 33, o fim da primeira linha e o ponteiro destino (DE) com D-FILE; assim:


```
MEM. 31ØØØ  LD    DE,(16396)  'ED5BØC4Ø'  ; carrega DE com D-FILE
MEM. 31ØØ4  LD    HL,(16396)  '2AØC4Ø'    ; idem para HL
MEM. 31ØØ7  LD    A,L         '7D'
MEM. 31ØØ8  ADD   A,33        'C621'      ; soma 33 a HL
MEM. 31Ø1Ø  LD    L,A         '6F'
MEM. 31Ø11  LD    A,H         '7C'
MEM. 31Ø12  ADC   A,Ø         'CEØØ'      ; transfere o "vai um"
                                            (CARRY)
MEM. 31Ø14  LD    H,A         '67'
MEM. 31Ø15  LD    BC,693      'Ø1B5Ø2'    ; carrega contador
MEM. 31Ø18  LDIR              'EDBØ'      ; loop de transferência
MEM. 31Ø2Ø  RET               'C9'
```

Prog. 4.6 – SCROLL.

Note a maneira como 33 foi somado a HL. Isto porque não existe instrução para somar dados *diretamente* a pares de registros (ou seja, não existe a instrução ADD HL,33).

Coloque este programa a partir da memória 31ØØØ (basta fazer memória inicial = 31ØØØ no programa HEXAMEM).

Agora execute:

```
2990 SLOW
3000 FOR I=1 TO 100
3010 PRINT AT 21,USR 31000;"                                "
3020 PRINT AT 21,0;I
3030 NEXT I
```

Prog. 4.7 – Exemplo de SCROLL.

Vamos apresentar agora um programa bastante interessante em BASIC que utiliza estas duas sub-rotinas (SCROLL e ANTISCROLL) e que mostra a aplicação das mesmas para "rolar" a tela, deslocando-a para cima ou para baixo apertando as teclas 6 e 7, sendo a tela composta por caracteres gerados aleatoriamente. Mais tarde, você poderá fazer jogos gráficos emocionantes usando estas sub-rotinas; basta usar a imaginação...

```
3000 FAST
3005 RAND
3010 DIM A$(22,32)
3015 FOR I=1 TO 22
3020 LET B$=CHR$ (63*RND+128*(RN
D>.5))
3025 FOR J=1 TO 5
3030 LET B$=B$+B$
3035 NEXT J
3040 LET A$(I)=B$
3045 PRINT A$(I)
3050 NEXT I
3055 LET A=1
3060 SLOW
3065 IF INKEY$="" THEN GOTO 3065
3070 LET B=A+1
3075 IF B=23 THEN LET B=1
3080 LET C=A-1
3085 IF C=0 THEN LET C=22
3090 LET B$=INKEY$
3095 IF B$="6" THEN PRINT AT 0,U
SR 30000;A$(C)
4000 IF B$="7" THEN PRINT AT 21,
USR 31000;A$(B)
4005 IF B$="6" THEN LET A=C
4010 IF B$="7" THEN LET A=B
4015 GOTO 3055
```

Prog. 4.8 — "Rolando" a tela...

Execute novamente o programa, substituindo a linha 3020 por

3020 LET B$ = CHR$ (128 + INT (11 * RND))

Vale a pena notar a velocidade da linguagem de máquina: cada vez que o SCROLL ou ANTISCROLL são chamados, toda a tela é deslocada (693 bytes) em décimos de segundo...

Resumo

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

Vimos um programa para fazer o ANTISCROLL onde aprendemos mais uma variável do programa interpretador, chamada D-FILE, que está nos endereços 16396 e 16397 e contém a posição na memória (menos um) do primeiro caractere da tela.

Exercícios

1. Faça um programa para fazer um SCROLL apenas da metade *superior* da tela, deixando o resto inalterado. Coloque-o a partir da memória 30000.

2. Faça um programa para fazer um ANTISCROLL apenas da metade *inferior* da tela, deixando o resto inalterado. Coloque-o a partir da memória 31000.

   Sugestão: Use o seguinte programa para testar suas sub-rotinas:

```
5995 SLOW
6000 FOR I=1 TO 22
6005 PRINT I
6010 NEXT I
6015 FOR I=1 TO 11
6020 PRINT AT 10,USR 30000;"    "
6025 PRINT AT 10,0;I;" SCROLL ME
TADE SUPERIOR"
6030 NEXT I
6035 FOR I=22 TO 12 STEP -1
6040 PRINT AT 11,USR 31000;"    "
6045 PRINT AT 11,0;I;" ANTI-SCRO
LL METADE INFERIOR"
6050 NEXT I
```

Prog. 4.9 — Programa teste para SCROLL e ANTISCROLL de meia tela.

3. Analise o funcionamento de *todos* os programas em BASIC que aparecem neste capítulo.

Instruções vistas neste capítulo:

LDI
LDIR
LDD
LDDR

O STACK POINTER e o PROGRAM COUNTER: Instruções de Salto, Sub-rotinas e Números Negativos

DOIS REGISTROS DE 16 BITS: O SP E O PC

No capítulo 3 falamos sobre a flag de CARRY que fazia parte de um registro interno do microprocessador chamado F. Vamos estudar agora, 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.

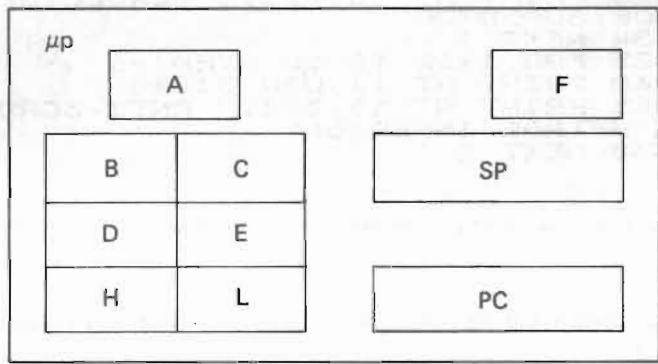


Fig. 5.1 – Como podemos imaginar os registros internos do microprocessador.

O STACK POINTER: 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 desta "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 contém o número 12345 (H = '3Ø' e L = '39') e que o registro SP contém o número 3Ø1ØØ.

A instrução PUSH HL coloca o conteúdo do registro H na memória de endereço 3ØØ99 (SP-1) e o registro L na memória 3ØØ98 (SP-2). Com isso, o registro SP passa a indicar 3ØØ98. 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 3Ø1ØØ. Estas 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 = 3Ø1ØØ), se em vez de PUSH HL tivéssemos feito POP HL, o conteúdo do par HL seria perdido e substituído pelo que estivesse nas memórias 3Ø1ØØ (registro L) e 3Ø1Ø1 (registro H), passando SP a indicar 3Ø1Ø2.

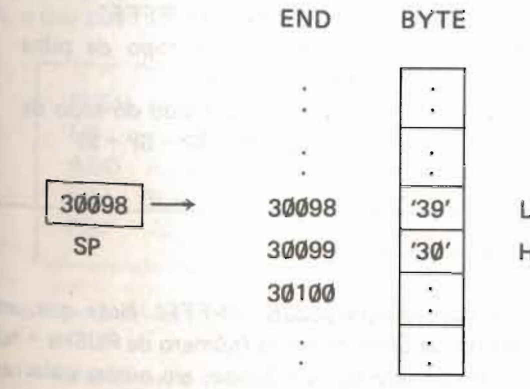


Fig. 5.2 – Resultado da operação PUSH HL, estando SP inicialmente indicando 3Ø1ØØ.

Assim como nos loops de 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).

Experimente executar os seguintes programas, que colocam 'FFFF' em BC, sendo que o primeiro *troca* os conteúdos de HL e BC e o segundo *copia* o conteúdo de HL em BC:

```
30000  LD    HL,65535   '21FFFF'   ; carrega HL com 'FFFF'
30003  PUSH  HL         'E5'       ; coloca HL no topo da pilha
                                     (SP = SP – 2)
30004  PUSH  BC         'C5'       ; coloca BC no topo da pilha
                                     (SP = SP – 2)
30005  POP   HL         'E1'       ; coloca o conteúdo do topo da
                                     pilha em HL (SP = SP + 2)
30006  POP   BC         'C1'       ; coloca o conteúdo do topo da
                                     pilha em BC (SP = SP + 2)
30007  RET              'C9'
```

Prog. 5.1 – Uso de PUSH e POP para trocar o conteúdo de pares de registros.

```
30000 LD    HL,65535   '21FFFF'   ; carrega HL com 'FFFF'
30003 PUSH  HL         'E5'       ; coloca HL no topo da pilha
                                    (SP = SP – 2)
30004 POP   BC         'C1'       ; coloca o conteúdo do topo da
                                    pilha em BC (SP = SP + 2)
30005 RET              'C9'
```

Prog. 5.2 – Uso de PUSH e POP para copiar HL em BC.

Execute os dois programas e você obterá 65535 = 'FFFF'. Note que, em ambos os casos, os valores inicial e final de SP eram iguais (número de PUSHs = número de POPs). É importante que isso *sempre* seja verificado; em outras palavras, o valor de SP no início de um programa deve ser igual ao seu valor no fim do mesmo. Com o tempo você perceberá o porquê disso.

O STACK POINTER deve ter como seu valor inicial um endereço bem no fim da memória para evitar que os dados empilhados atinjam os programas. Assim, por exemplo, se SP = 30100 e tivermos um programa que termina em 30060, com mais de 20 PUSHs antes que haja algum 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 conseqüências não desejáveis... (um belo quadro de arte moderna, por exemplo). O valor inicial do SP fica a cargo do programa interpretador e está nas memórias 16386 e 16387, numa variável chamada ERR-SP; assim, essa variável indica sempre a parte de baixo da pilha, enquanto SP indica o topo.

Aqui estão os códigos das instruções de PUSH e POP:

INSTRUÇÃO	CÓDIGO	INSTRUÇÃO	CÓDIGO
PUSH AF	'F5'	POP AF	'F1'
PUSH BC	'C5'	POP BC	'C1'
PUSH DE	'D5'	POP DE	'D1'
PUSH HL	'E5'	POP HL	'E1'

Tab. 5.1 – Instruções PUSH e POP.

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 ao registro B 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:

```
PUSH AF    'F5'    ; "salva" AF no topo da pilha
LD   A,B   '78'    ; coloca B em A (destrói o antigo valor de A)
ADD  A,25  'C619'  ; soma 25
LD   B,A   '47'    ; coloca A em B
POP  AF    'F1'    ; retorna AF do topo da pilha
```

Prog. 5.3 – Soma sem alterar o acumulador e a flag de CARRY.

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:

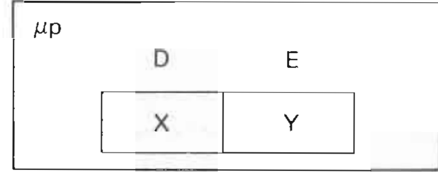
INSTRUÇÃO	CÓDIGO
LD SP,HL	'F9' (esta instrução permite a cópia direta de 16 bits!)
LD SP,dado	'31' + 2 bytes para o dado
LD SP,(endereço)	'ED7B' + 2 bytes para endereço
LD (endereço), SP	'ED73' + 2 bytes para endereço
ADD HL,SP	'39'
ADC HL,SP	'ED7A'
SBC HL,SP	'ED72'
INC SP	'33'
DEC SP	'3B'

Tab. 5.2 – Instruções que utilizam diretamente o SP.

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:

```
PUSH   DE   'D5'
PUSH   DE   'D5'
INC    SP   '33'
POP    DE   'D1'
INC    SP   '33'
```

Prog. 5.4 – Permuta de D e E.



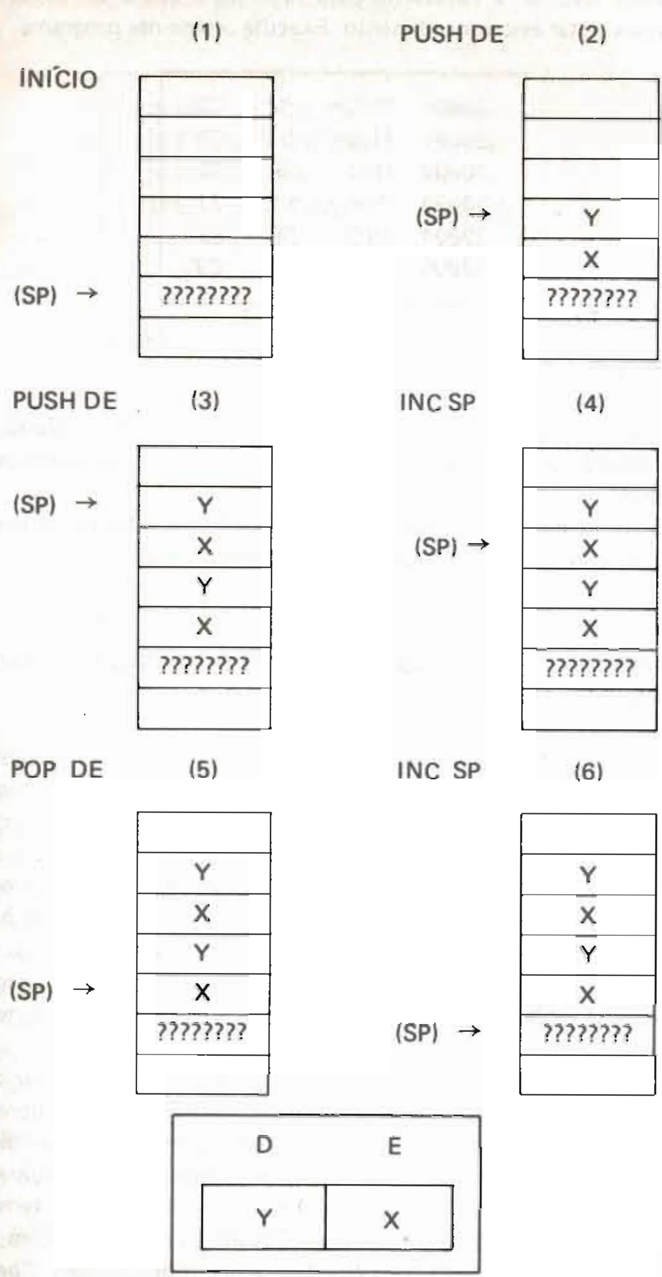


Fig. 5.3 – Visualização dos registros DE no micro e da pilha na memória em passos sucessivos.

O último INC SP é necessário para fazer com que o SP *volte a seu valor original*. Vamos testar esse procedimento. Execute o seguinte programa:

```
30000  PUSH  BC  'C5'
30001  PUSH  BC  'C5'
30002  INC   SP  '33'
30003  POP   BC  'C1'
30004  INC   SP  '33'
30005  RET       'C9'
```

Prog. 5.5 — Permuta de B e C.

Como esse programa está colocado a partir da memória 30000, ao executá-lo (USR 30000) o par BC é carregado com '7530' e no final devemos ter '3075', ou seja, 12405.

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

O PROGRAM COUNTER E AS INSTRUÇÕES DE SALTO ABSOLUTOS E RELATIVOS

O PC (Program Counter = contador de programa) é outro registro interno de 16 bits que *não* pode ser dividido e cuja função é indicar ao microprocessador em que endereço da memória ele deve "buscar" a instrução em linguagem de máquina. Assim, ao fazer USR 30000, os pares BC e PC são carregados com 30000 e o microprocessador executa a instrução na memória 30000 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 1 byte, ela será executada e PC passará a indicar 30001; se a segunda instrução for LD A,2 (2 bytes), ela será executada e PC indicará 30003, 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 30000 carregará PC com o endereço 30000 (sem, no entanto, incrementá-lo) e o microprocessador "pensará" que 30000 é o endereço da próxima instrução do programa.

Note que se tivermos um programa do tipo:

```
30000  LD   A,2     '3E02'
30002  LD   HL,345  '215901'
30005  INC  A       '3C'
30006  JP   30003   'C33375'
```

Prog. 5.6 — Exemplo de uso *errado* da instrução JP.

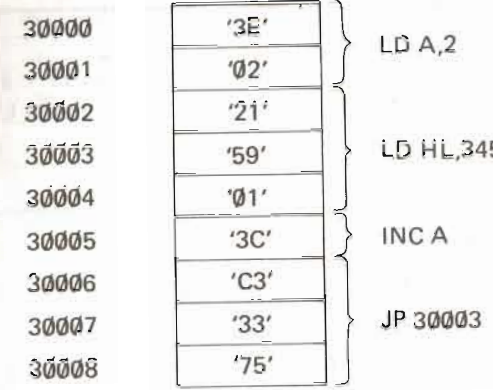


Fig. 5.4 — Visualização de memória.

é provável que "coisas estranhas" aconteçam, pois estamos indo com JP 30003 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 30003 (cujo conteúdo é '59') como o começo de uma instrução; o que certamente causará "coisas malucas"!

Ainda usando o exemplo anterior, vamos substituir o nosso JUMP por JP 30000. Veja que, apesar de agora esse endereço fazer sentido, acabamos de criar um *loop infinito*, de onde o TK nunca sairá, nem mesmo usando BREAK! O único remédio é... desligar o computador.

O código da instrução JUMP é:

JP endereço 'C3' + 2 bytes para endereço

ou, usando o endereço dado pelo par HL:

JP (HL) código 'E9'

Note que agora é fácil notar porque, além de colocarmos os mnemônicos e seus códigos, devemos colocar também os endereços do começo de cada instrução,

pois eles, além de mostrar *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 Ø 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:

```
30000   INC  A
30001   JR   2          ; próximo byte 30003 + 2 = 30005 que
                          cai no meio da instrução LD HL, 64000!
30003   LD   HL,64000
30006   DEC  A
```

Prog. 5.7 — Exemplo de utilização *errada* de JR.

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

Surge agora uma pergunta: e para "pular" para trás? Devemos usar números negativos? Como representá-los só com Øs e 1s? 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 Ø 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: "ØØØØ ØØØØ"
- números positivos: de "ØØØØ ØØØ1" (número + 1) 'Ø1'
  até "Ø111 1111" (número + 127) '7F'
- números negativos: pegue o número positivo correspondente em binário, troque os Øs pelos 1s 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 = "ØØØØ ØØ1Ø" ⇒ "1111 11Ø1"
  + "1"
  "1111 111Ø"
  ∴ −2 = 'FE'

De qualquer forma, para facilitar vamos apresentar uma tabela com os números negativos de −1 a −128.

	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	-128	-127	-126	-125	-124	-123	-122	-121	-12Ø	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-11Ø	-1Ø9	-1Ø8	-1Ø7	-1Ø6	-1Ø5	-1Ø4	-1Ø3	-1Ø2	-1Ø1	-1ØØ	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-9Ø	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-8Ø	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-6Ø	-59	-58	-57	-56	-55	-54	-53	-52	-51	-5Ø	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-4Ø	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-3Ø	-29	-28	-27	-26	-25	-24	-23	-22	-21	-2Ø	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-1Ø	-9	-8	-7	-6	-5	-4	-3	-2	-1

Tab. 5.3 — Números negativos de 1 byte.

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

−55 = 'C9' = +2Ø1
−78 = 'B2' = +178

Você pode também obter a representação dos números negativos da seguinte forma (RUN 7Ø5):

```
7Ø5 SLOW
71Ø FOR I=-1 TO -128 STEP -1
72Ø POKE 30000,I
73Ø LET J=PEEK 30000
74Ø SCROLL
75Ø PRINT I,J
76Ø NEXT I
```

Prog. 5.8 — Programa para obter a representação decimal dos números negativos.

Você saberia explicar por quê? Note que, utilizando esta convenção, para saber se um número é positivo ou negativo basta olhar o seu bit mais significativo (bit 7 para números de 1 byte): se ele for 1, o número é negativo, caso contrário, é positivo.

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 'FE') significa fazer um loop infinito, 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 na memória 3ØØØ2 e deseja voltar para a memória 3ØØØØ; então:

3ØØØ2 JR –4 '18FC'

OS SALTOS CONDICIONAIS BASEADOS NAS FLAGS

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: os saltos *condicionais*, ou seja, "salte se ocorrer dada condição". 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 ('38' + 1 byte para dado)
(Jump Relative if Carry = 1)
JR NC,dado ('3Ø' + 1 byte para dado)
(Jump Relative if No Carry)
JP C,endereço ('DA' + 2 bytes para endereço)
JP NC,endereço ('D2' + 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 Øs do *acumulador* é par (EVEN) ou ímpar (ODD) *após efetuada uma operação lógica* (ver apêndice 2). Assim, temos:

JP PE,endereço ('EA' + 2 bytes para endereço)
(JumP if Parity is Even)
JP PO,endereço ('E2' + 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 negati-

vos, podem haver trocas acidentais de sinais; por exemplo, se você somar '4A', que é positivo, com '44', que também é positivo, você obterá '8E', que é negativo, pois agora o maior número positivo representável em 1 byte é 127 ('7F'). Note que você terá OVERFLOW mas *não* CARRY = 1 pois, usando a convenção de números apenas positivos, esta soma produz CARRY = Ø. 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). (Ver apêndice 2.)

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

JR Z,dado ('28' + 1 byte para dado)
(Jump Relative if Zero)
JR NZ,dado ('2Ø' + 1 byte para dado)
(Jump Relative if Not Zero)
JP Z,endereço ('CA' + 2 bytes para endereço)
JP NZ,endereço ('C2' + 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 ('FA' + 2 bytes para endereço)
(JumP if Minus)
JP P,endereço ('F2' + 2 bytes para endereço)
(JumP if Plus)

Também *não* existe salto relativo para esta condição.

As flags ensinadas até agora estão localizadas da seguinte maneira no registro F:

BIT	7	6	5	4	3	2	1	Ø
	S	Z				P/O		C

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

Fig. 5.5 – As flags no registro F.

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, explicitar 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 na memória 3ØØØØ e quiser colocá-lo a partir da memória 3Ø1ØØ, todos os endereços dos JUMPs terão de ser acrescidos de 1ØØ. Isto *não* acontece com os JUMPs relativos (JR), fato este que os torna facilmente relocáveis.

Apenas para complementar as instruções com números negativos, apresentaremos a instrução NEG, que muda o sinal do número que está no acumulador (seu código é 'ED44'). 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 seguinte programa:

```
3ØØØØ   LD    B,A    '47'     ; coloca A em B
3ØØØ1   NEG          'ED44'   ; ou NEG A
3ØØØ3   ADD   A,B    '8Ø'     ; soma A com B
3ØØØ4   LD    B,A    '47'     ; prepara a saída
3ØØØ5   LD    C,A    '4F'
3ØØØ6   RET          'C9'
```

Prog. 5.9 – Exemplo de uso de NEG.

Você logicamente irá obter zero! Essa instrução altera *todas* as flags (ver apêndice 2).

Como você deve ter notado, nas instruções de salto condicional nós sublinhamos a frase: *resultado de uma operação*. De fato, os JUMPs condicionais funcionam se, antes, alguma operação afetou as flags *adequadamente*. Por exemplo, não adianta carregar o acumulador com o conteúdo de algum registro ou memória e, em seguida, querer usar um JUMP condicional baseado no conteúdo do acumulador. Isso simplesmente utilizará para testes as flags da *última* operação que foi efetuada antes desse carregamento. Logo, se você quer "saltar" para algum ponto (3ØØ2Ø), baseado, por exemplo, no fato de uma determinada memória (3ØØ1Ø) conter o número zero, não adianta fazer o seguinte:


```
LD   A,(3ØØ1Ø)   ; não afeta nenhuma flag
JP   Z,3Ø12Ø     ; testa a flag gerada na última operação
```

Prog. 5.1Ø – Exemplo de como não usar JP condicional após LD.

Isso é válido para *todos* os saltos condicionais. Para poder testar condições, nesses casos, é necessária então uma nova instrução, chamada CP (ComPare), que compara o conteúdo do acumulador com qualquer outro registro, com um dado número (entre Ø e 255 ou entre −128 e 127) ou com a memória endereçada 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* (Ø) 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. (Veja exercício 13.)

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

```
LD   A,(3ØØ1Ø)            ; coloca em A o conteúdo da memória
                            3ØØ1Ø
CP   Ø            'FEØØ'  ; compara A com o número Ø: se for
                            igual, flag Z = 1; caso contrário, flag
                            Z = Ø
JP   Z,3Ø12Ø              ; pula para 3Ø12Ø se flag Z = 1; caso con-
                            trário, prossegue (JP M seria usado para
                            testar se A < Ø e JP P para testar se
                            A > Ø)
```

Prog. 5.11 – Exemplo de uso correto de JP condicional após LD.

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. Eis os códigos:

INSTRUÇÃO		CÓDIGO
CP	dado	'FE' + 1 byte para o dado
CP	(HL)	'BE'
CP	A	'BF' → (Você vê alguma utilidade para esta instrução?)
CP	B	'B8'
CP	C	'B9'
CP	D	'BA'
CP	E	'BB'
CP	H	'BC'
CP	L	'BD'

Tab. 5.4 – A instrução CP.

Aqui, vale novamente a observação sobre possibilidade de analogia:

```
CP  '3B'  com  CP  A,'3B'
CP  (HL)  com  CP  A,HL
CP  A     com  CP  A,A
CP  D     com  CP  A,D
```

No entanto, é bom lembrar que os programas ASSEMBLER reconhecem apenas os códigos convencionais.

COMPLETANDO O ANTISCROLL: LOOPS EM LINGUAGEM DE MÁQUINA

Vamos, a título de exemplo, completar o programa de ANTISCROLL apresentado no capítulo anterior, ou seja, vamos fazer em linguagem de máquina o que a instrução PRINT "32 brancos" faz. Para isto, é necessário notar que as instruções de DEC *afetam* a flag de *zero* (Z) (ver apêndice 2), ou seja, se ao decrementar dado registro, ou memória, o resultado for *zero*, isto será indicado na flag Z!

```
30000  LD    BC,726        '01D602'
30003  LD    HL,(16396)    '2A0C40'
30006  ADD   HL,BC         '09'
30007  LD    D,H           '54'
30008  LD    E,L           '5D'
30009  LD    BC,693        '01B502'
30012  LD    HL,(16396)    '2A0C40'
30015  ADD   HL,BC         '09'
30016  LDDR                'EDB8'
30018  LD    DE,32         '112000'   ; carrega DE com 32
30021  ADD   HL,DE         '19'       ; coloca em HL o endereço da
                                        última posição da primeira
                                        linha da tela
30022  LD    (HL),0        '3600'     ; coloca espaço em branco (có-
                                        digo 0) na posição indicada
                                        por HL
30024  DEC   HL            '2B'       ; decrementa HL
30025  DEC   E             '1D'       ; decrementa E, que é utilizado
                                        como contador
30026  JP    NZ,30022      'C24675'
   ou (JR    NZ,-6         '20FA')    ; se E não for zero, volta para
                                        colocar outro espaço em branco
30029  RET                 'C9'
```

Prog. 5.12 – ANTISCROLL completo.

Alexandre Fernandes
RG: 14.317.733

Como você pode verificar, esse procedimento coloca o número zero, que é o código de espaço em branco para o TK, nas memórias que correspondem à primeira linha da tela. Após o LDDR, o par HL indica novamente o começo da tela; assim, basta somar 32 para acharmos o endereço do último caractere da primeira linha. Ao carregar DE com 32, teremos D = '00' e E = '20' ('20' = 32), e poderemos então usar o registro E como contador para colocar os "32 brancos" necessários na memória. Ao colocar o último branco, DEC E fará E = '00', colocando a flag Z em 1, e o programa não mais saltará para o endereço 30022, voltando então para o BASIC. Note que o par BC continua indicando '0000'. Assim, como você pode notar, fizemos um *loop com linguagem de máquina* utilizando a instrução DEC e o salto condicional baseado na flag de zero gerada.

Vamos testar o programa:

```
1005 SLOW
1010 FOR I=1 TO 80
1020 RAND USR 30000
1030 PRINT AT 0,0;TAB I;I
1040 NEXT I
```

Prog. 5.13 – Exemplo de uso de ANTISCROLL.

Observação: Teste a sub-rotina em linguagem de máquina usando JP NZ 30022 e, depois, JR NZ,-6.

Note que não nos interessa a saída da sub-rotina em linguagem de máquina (ou seja, par BC), mas precisamos chamá-la para que ela seja executada. Utilizamos então a instrução RAND que, no caso, não faz nada: apenas permite que o cursor mude para F *após uma key-word* para colocar USR pois, sem isso, a linha não é aceita no programa BASIC (tente colocar no programa a linha 2Ø USR 3ØØØØ).

Existe em linguagem de máquina uma instrução especial que facilita executar loops:

DJNZ dado ('1Ø' + 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.

Assim, o final do programa anterior poderia ser:

```
3ØØ22  LD   B,E      '43'     ; coloca o contador em B
3ØØ23  LD   (HL),Ø   '36ØØ'
3ØØ25  DEC  HL       '2B'
3ØØ26  DJNZ -5       '1ØFB'   ; decrementa o contador B e "salta" de
                                volta se ele não for zero
3ØØ28  RET           'C9'
```

Prog. 5.14 – Refinando o final do ANTISCROLL.

Teste então esta terceira possibilidade.

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 Ø 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 da tradução dos mnemônicos, pois não constam nas tabelas, além de não funcionarem com os programas tradutores ASSEMBLERs. No nosso caso, a escolha fica a seu critério.

O STACK POINTER: SUB-ROTINAS

Mesmo em linguagem de máquina podemos ter sub-rotinas; temos então uma instrução equivalente ao GOSUB que seria:

CALL endereço ('CD' + 2 bytes para endereço)

e, para retornar ao programa principal, temos a já famosa RET ('C9'). 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:

INSTRUÇÃO	CÓDIGO
CALL Z , endereço	'CC' + 2 bytes para endereço
CALL NZ , endereço	'C4' + 2 bytes para endereço
CALL C , endereço	'DC' + 2 bytes para endereço
CALL NC , endereço	'D4' + 2 bytes para endereço
CALL PE , endereço	'EC' + 2 bytes para endereço
CALL PO , endereço	'E4' + 2 bytes para endereço
CALL M , endereço	'FC' + 2 bytes para endereço
CALL P , endereço	'F4' + 2 bytes para endereço

Tab. 5.5 – A instrução CALL.

e, como era de se esperar, podemos também fazer RET condicionalmente:

INSTRUÇÃO		CÓDIGO
RET	Z	'C8'
RET	NZ	'CØ'
RET	C	'D8'
RET	NC	'DØ'
RET	PE	'E8'
RET	PO	'EØ'
RET	M	'F8'
RET	P	'FØ'

Tab. 5.6 – A instrução RET.

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 a partir desse endereço e, ao encontrar uma instrução 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 da memória 30100, uma sub-rotina para fazer uma soma de dois registros da memória indicados por DE e HL:

```
30100  LD   A,(DE)
30101  LD   B,A
30102  LD   A,(HL)
30103  ADD  A,B
30104  RET
```

Prog. 5.15 – Sub-rotina para somar dois conteúdos de memória.

Essa sub-rotina poderia ser chamada pelo seguinte programa:

```
30000  LD    HL,30150
30003  LD    DE,30155
30006  CALL  30100
30009  INC   HL
30010  INC   DE
30011  LD    (30160),A
30014  CALL  30100
30017  INC   HL
30018  INC   DE
30019  LD    (30161),A
30022  RET
```

Prog. 5.16 – Programa que chama a sub-rotina anterior.

Ao fazer o primeiro CALL 30100, teremos no topo da pilha o valor 30009 ('7539'), que corresponde ao endereço de retorno da sub-rotina, e teremos também PC = 30100. O microprocessador passa a executar o programa a partir da memória 30100, e, ao encontrar o RET, o PC é carregado com o topo da pilha; assim PC = 30009, que corresponde à primeira instrução INC HL. O "andamento" do SP é mostrado na figura 5.6. No segundo CALL, o topo da pilha indicará agora 30017 ('7541'), e o processo se repete.

Cuidados devem ser tomados sempre que, durante uma sub-rotina, mexermos no topo da pilha, para evitar que o programa, ao encontrar um RET, volte para posições não desejáveis (pois o endereço de retorno da sub-rotina é colocado no topo da pilha pela instrução CALL). No exemplo anterior fizemos CALL 30100 (= CALL '7594'); antes do CALL tínhamos SP = SP_1 (ver figura 5.6) e, após o CALL, SP = SP_2.

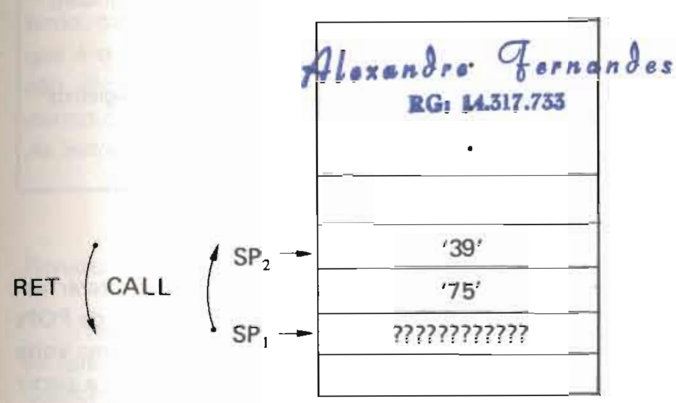


Fig. 5.6 – Visualização da pilha na memória RAM.

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 PUSH. 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 30100 e tente colocar os códigos em linguagem de máquina (utilize o apêndice 2 para ajudá-lo).

```
30100  PUSH  BC     ; "salva" B e C
30101  LD    B,E    ; coloca um dos operandos no contador B
30102  PUSH  DE     ; "salva" D e E
30103  LD    D,0    ; coloca o outro operando no par DE
30105  LD    E,A
30106  LD    HL,0   ; coloca 0 no par HL
30109  ADD   HL,DE  ; soma o segundo operando sucessivamente
                      tantas vezes quantas indicar o primeiro
                      operando (B)
30110  DJNZ  -3
30112  POP   DE     ; reestabelece os valores iniciais dos registros
30113  POP   BC
30114  RET
```

Prog. 5.17 – Sub-rotina que multiplica números.

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 da memória **30000**, que multiplique dois números (por exemplo, 15 e 24), usando esta sub-rotina:

```
30000  LD    A,15   ; carrega operandos
30002  LD    E,24
30004  CALL  30100  ; chama sub-rotina de multiplicação (não
                      esqueça da inversão!)
30007  LD    B,H    ; prepara a saída
30008  LD    C,L
30009  RET
```

Prog. 5.18 – Programa *principal* que chama a sub-rotina (prog. 5.17).

Verifique o resultado (que deve ser 360) e execute o programa usando XF no HEXAMEM. Aliás, quando não temos nenhum efeito especial de tela mas apenas cálculos, é conveniente executar os programas em FAST.

A sub-rotina aqui utilizada é bastante simples mas, se usarmos alguma mais complexa, poderemos eventualmente esquecer de fazer o SP voltar a seu valor original... Portanto, eis um conselho: existem dois endereço na RAM (16507 e 16508) que *nunca* são usados pelo computador; assim, é aconselhável que sem-

pre que for usar uma sub-rotina complexa, você "salve" logo no começo o valor do SP:

LD (16507),SP (que coloca SP nos endereços 16507 e 16508)

e, no fim da sub-rotina:

LD SP,(16507)

Dessa forma, estará garantida sua volta ao lugar certo do programa. Note, entretanto, que você está "salvando" o valor de SP e *não* o conteúdo do topo da pilha, que é o endereço de retorno. Assim, você pode alterar à vontade SP, desde que não altere as duas posições de memória que indicam o endereço de retorno (por exemplo, fazendo INC SP e, a seguir, PUSH BC. Você saberia explicar por quê?). As instruções CALL e RET *não* afetam nenhuma flag.

Resumo

Neste capítulo vimos dois novos registros internos do microprocessador 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 com o 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), PARITY/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.

Finalmente, vimos como chamar uma sub-rotina em linguagem de máquina quando não nos interessa sua saída (RAND USR endereço) e vimos duas novas regiões da RAM: 16507 e 16508, que são bytes livres onde poderemos armazenar o que quisermos (no caso, elas foram indicadas para guardar o valor do SP ao fazer uma sub-rotina complexa).

Exercícios

1. Faça um programa que execute um loop utilizando a instrução INC. Por exemplo, mande escrever na tela os primeiros 20 caracteres do TK. Lembre-se que o endereço da primeira posição da tela está indicado nas memórias 16396 e 16397, devendo-se somar 1 pois o primeiro código é de NEW LINE.
   Observação: Você deverá usar a instrução CP para verificar o fim do loop.
   Sugestão: Observe o programa de ANTISCROLL apresentado neste capítulo.
2. Faça um programa que "varra" uma dada região de memória (por exemplo, os 60 bytes após a memória 20000), coloque todos os números que forem menores que 66 ('42') 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 memórias sucessivas (RAM) e elabore um programa em BASIC que faça um PEEK dessas seis memórias.
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 nas memórias 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 uma sub-rotina em linguagem de máquina que some dois registros e detecte se houve ou não CARRY. Se houve CARRY, ela deve colocar o caractere C (código 40 ou '28') na tela na primeira coluna da segunda linha; caso contrário, ela deverá colocar o caractere N (código 51 ou '33') (lembre-se de D-FILE).
8. Você saberia, usando quatro instruções, colocar o *valor* de SP nos registros B e C? Feito isso, acrescente um RET e execute o programa para obter o valor de SP na tela.
9. Você saberia colocar o registro F no registro C? Feito isso, "zere" o registro B e acrescente um RET, executando o programa para obter o valor desse registro na tela.
10. Complemente o exercício 2 de maneira que o programa coloque na pilha todos os números que forem *menores ou iguais* a 130 e *maiores que* 10. Faça com que a saída do programa na tela seja a quantidade de números que ele achou dentro desse intervalo.
    Você deve ter usado as memórias 16507 e 16508 para colocar o valor de SP e poder retornar sem problemas ao BASIC. Assim, para verificar se sua sub-rotina funcionou de acordo, basta fazer um PEEK das memórias *acima* da memória indicada por SP tantas vezes quantas forem indicadas pela saída do programa do exercício 10. Suponha que você obteve 15 números no intervalo mencionado; assim:

```
  10 LET SP=PEEK 16507+256*PEEK
16508
  20 FOR I=1 TO 15
  30 LET SP=SP-1
  40 SCROLL
  50 PRINT "MEMORIA ";SP;" CONTE
UDO",PEEK SP
  60 NEXT I
```

Tente explicar esse programa.

11. Por que não é permitido fazer instruções de PUSH e POP utilizando registros simples? Poderia ser utilizada (ou "criada") uma instrução para colocar 3 registros no topo da pilha? E 4 registros? O que isto implicaria se utilizássemos essas instruções "dentro" de uma sub-rotina?
12. Analise cuidadosamente as instruções da tabela 5.2 e procure utilizá-las em seus programas.
13. Neste volume, não vamos dar muita ênfase aos números negativos. Entretanto, para "meditação", verifique se a explicação, dada na página 85, sobre desigualdade de números, vale para números maiores que 128, ou seja, usando a convenção de números só positivos. E, no caso de usar a outra convenção, apenas o teste da flag de sinal é suficiente em todos os casos? (A resposta a essa pergunta é muito importante e ela estará detalhada no próximo volume.)

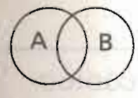
Instruções vistas neste capítulo

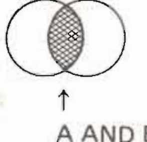
PUSH	par de registros	incluindo AF
POP	par de registros	
LD	SP,HL	
LD	SP,dado	
LD	SP,(endereço)	
LD	(endereço),SP	
ADD	HL,SP	
ADC	HL,SP	
SBC	HL,SP	
INC	SP	
DEC	SP	
JP	endereço	
JP	(condições: C, NC, PO, PE, Z, NZ, M, P) ,endereço	
JR	dado (−128 a +127)	
JR	(condições: C, NC, Z, NZ) ,dado	
JP	(HL)	
NEG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

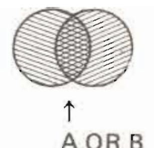
Instruções Lógicas e Operações com bits

AS OPERAÇÕES LÓGICAS AND, OR, NOT e XOR

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 Ø, 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.







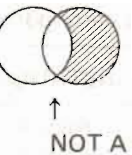


Fig. 6.1 — Diagramas para visualizar as operações AND, OR e NOT envolvendo os conjuntos A e B.

Logo:

1 AND 1 = 1	1 OR 1 = 1	NOT Ø = 1
1 AND Ø = Ø AND 1 = Ø	1 OR Ø = Ø OR 1 = 1	NOT 1 = Ø
Ø AND Ø = Ø	Ø OR Ø = Ø	

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 = 'C7' e do registro, B = 'A9'; falando em bits, temos:

A = "11ØØØ111"
B = "1Ø1Ø1ØØ1"

assim:

A AND B = "10000001" = '81'
A OR B = "11101111" = 'EF'
NOT A = "00111000" = '38'

Observação: Colocamos sempre os números binários entre aspas (" ").

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 = "01101110" = '6E'

Seguem-se as seguintes instruções em linguagem de máquina:

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 a:

INSTRUÇÃO	CÓDIGO	INSTRUÇÃO	CÓDIGO	INSTRUÇÃO	CÓDIGO
AND A	'A7'	OR A	'B7'	XOR A	'AF'
AND B	'A0'	OR B	'B0'	XOR B	'A8'
AND C	'A1'	OR C	'B1'	XOR C	'A9'
AND D	'A2'	OR D	'B2'	XOR D	'AA'
AND E	'A3'	OR E	'B3'	XOR E	'AB'
AND H	'A4'	OR H	'B4'	XOR H	'AC'
AND L	'A5'	OR L	'B5'	XOR L	'AD'

Tab. 6.1 — Instruções AND, OR e XOR.

Essas três instruções afetam a flag do CARRY (deixando-a em 0 (zero)) 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 '2F')

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 programas ASSEMBLER), o mesmo ocorrendo para as instruções NEG, CP etc. Entretanto, poderia surgir a pergunta: então por que 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.

Observação: Se você quiser usar freqüentemente essas instruções, seria conveniente fazer um pequeno programa em BASIC para converter números decimais e hexadecimais em binários e vice-versa.

AND

```
30000  LD   A,'AF'   '3EAF'
30002  LD   B,'5A'   '065A'
30004  AND  B        'A0'    ; (ou AND A,B)
30005  LD   C,A      '4F'
30006  LD   B,0      '0600'
30008  RET           'C9'
```

Prog. 6.1 — Instrução AND.

temos { A = "10101111" *resultando* A = "0000 1010" = '0A' = 10
B = "01011010"

OR

```
30000  LD   A,'82'   '3E82'
30002  LD   E,'27'   '1E27'
30004  OR   E        'B3'    ; (ou OR A,E)
30005  LD   C,A      '4F'
30006  LD   B,0      '0600'
30008  RET           'C9'
```

Prog. 6.2 — Instrução OR.

temos { A = "10000010" *resultando* A = "10100111" = 'A7' = 167
E = "00100111"

NOT

```
30000  LD    A,'F0'   '3EF0'
30002  CPL            '2F'     ; (ou CPL A)
30003  LD    C,A      '4F'
30004  XOR   A        'AF'
30005  LD    B,A      '47'
30006  RET            'C9'
```

Prog. 6.3 – Instrução CPL (NOT) e XOR.

temos: A = "11110000" *resultando* A = "00001111" = '0F' = 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:

INSTRUÇÃO	CÓDIGO
AND (HL)	'A6' ; (ou AND A,(HL))
OR (HL)	'B6' ; (ou OR A,(HL))
XOR (HL)	'AE' ; (ou XOR A,(HL))
AND dado	'E6' + 1 byte para o dado; (ou AND A,dado)
OR dado	'F6' + 1 byte para o dado; (ou OR A,dado)
XOR dado	'EE' + 1 byte para o dado; (ou XOR A,dado)

Tab. 6.2 – Instruções lógicas com as memórias e dados de 1 byte.

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 a memória 30050 se o conteúdo do registro B for positivo:

```
LD   A,B
OR   A           ; (ou OR A,A) não altera o acumulador, apenas as
                   flags (ver apêndice 2)
JP   P,  30050   ; "salta" para 30050 se positivo
```

Prog. 6.4 – Uso da instrução OR para alterar apenas as flags.

O mesmo poderia ter sido feito com AND A? E com XOR 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 uma tabela:

Bit \ Registro	A	B	C	D	E	H	L	(HL)
0	'CBC7'	'CBC0'	'CBC1'	'CBC2'	'CBC3'	'CBC4'	'CBC5'	'CBC6'
1	'CBCF'	'CBC8'	'CBC9'	'CBCA'	'CBCB'	'CBCC'	'CBCD'	'CBCE'
2	'CBD7'	'CDD0'	'CBD1'	'CBD2'	'CBD3'	'CBD4'	'CBD5'	'CBD6'
3	'CBDF'	'CBD8'	'CBD9'	'CBDA'	'CBDB'	'CBDC'	'CBDD'	'CBDE'
4	'CBE7'	'CBE0'	'CBE1'	'CBE2'	'CBE3'	'CBE4'	'CBE5'	'CBE6'
5	'CBEF'	'CBE8'	'CBE9'	'CBEA'	'CBEB'	'CBEC'	'CBED'	'CBEE'
6	'CBF7'	'CBF0'	'CBF1'	'CBF2'	'CBF3'	'CBF4'	'CBF5'	'CBF6'
7	'CBFF'	'CBF8'	'CBF9'	'CBFA'	'CBFB'	'CBFC'	'CBFD'	'CBFE'

Tab. 6.3 – SET.

Alexandre Fernandes
RG: 14.317.733

Bit \ Registro	A	B	C	D	E	H	L	(HL)
0	'CB87'	'CB80'	'CB81'	'CB82'	'CB83'	'CB84'	'CB85'	'CB86'
1	'CB8F'	'CB88'	'CB89'	'CB8A'	'CB8B'	'CB8C'	'CB8D'	'CB8E'
2	'CB97'	'CB90'	'CB91'	'CB92'	'CB93'	'CB94'	'CB95'	'CB96'
3	'CB9F'	'CB98'	'CB99'	'CB9A'	'CB9B'	'CB9C'	'CB9D'	'CB9E'
4	'CBA7'	'CBA0'	'CBA1'	'CBA2'	'CBA3'	'CBA4'	'CBA5'	'CBA6'
5	'CBAF'	'CBA8'	'CBA9'	'CBAA'	'CBAB'	'CBAC'	'CBAD'	'CBAE'
6	'CBB7'	'CBB0'	'CBB1'	'CBB2'	'CBB3'	'CBB4'	'CBB5'	'CBB6'
7	'CBBF'	'CBB8'	'CBB9'	'CBBA'	'CBBB'	'CBBC'	'CBBD'	'CBBE'

Tab. 6.4 – RESET.

Note que os oito bits de cada registro são "numerados" de Ø a 7. Assim, por exemplo, se você quiser colocar em 1 o bit 5 do registro E:

SET 5,E 'CBEB'

para colocar em Ø o bit 6 da memória indicada por HL:

RES 6,(HL) 'CBB6'

Façamos um exemplo: vamos chamar uma sub-rotina na memória 3ØØØØ, (teremos então BC = '753Ø' = "Ø111Ø1Ø1ØØ11ØØØØ") e vamos fazer B = 'ØØ' e C = 'FF' "setando" e "re-setando" os bits necessários, tendo então como saída (ao apertar XF no HEXAMEM) o número 255 ('ØØFF').

```
3ØØØØ   RES  6,B  'CBBØ'
3ØØØ2   RES  5,B  'CBA8'
3ØØØ4   RES  4,B  'CBAØ'
3ØØØ6   RES  2,B  'CB9Ø'
3ØØØ8   RES  Ø,B  'CB8Ø'
3ØØ1Ø   SET  7,C  'CBF9'
3ØØ12   SET  6,C  'CBF1'
3ØØ14   SET  3,C  'CBD9'
3ØØ16   SET  2,C  'CBD1'
3ØØ18   SET  1,C  'CBC9'
3ØØ2Ø   SET  Ø,C  'CBC1'
3ØØ22   RET       'C9'
```

Prog. 6.5 – Instruções SET e RES.

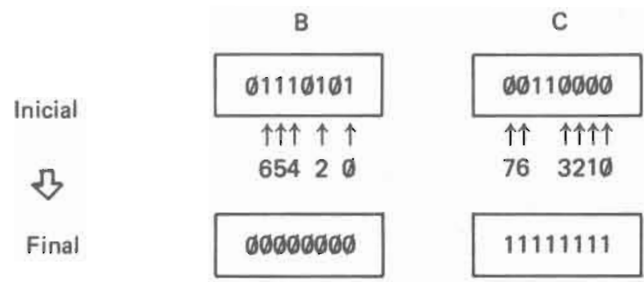


Fig. 6.2 – Registros B e C no início e fim do programa.

Se quisermos "testar" um determinado bit de um registro ou memória endereçada por HL para saber se ele é zero ou um, podemos usar a instrução BIT, que coloca o complemento do 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* (Ø), basta fazer:

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

Prog. 6.6 – A instrução BIT.

(Se a condição for que o bit seja 1, basta fazer JP NZ.)

Se o bit 5 é 1, o complemento é Ø (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:

Bit \ Registro	A	B	C	D	E	H	L	(HL)
Ø	'CB47'	'CB4Ø'	'CB41'	'CB42'	'CB43'	'CB44'	'CB45'	'CB46'
1	'CB4F'	'CB48'	'CB49'	'CB4A'	'CB4B'	'CB4C'	'CB4D'	'CB4E'
2	'CB57'	'CB5Ø'	'CB51'	'CB52'	'CB53'	'CB54'	'CB55'	'CB56'
3	'CB5F'	'CB58'	'CB59'	'CB5A'	'CB5B'	'CB5C'	'CB5D'	'CB5E'
4	'CB67'	'CB6Ø'	'CB61'	'CB62'	CB63'	'CB64'	'CB65'	'CB66'
5	'CB6F'	'CB68	'CB69'	'CB6A'	'CB6B'	'CB6C	'CB6D'	'CB6E'
6	'CB77'	'CB7Ø'	'CB71'	'CB72'	'CB73'	'CB74'	'CB75'	'CB76'
7	'CB7F'	'CB78'	'CB79'	'CB7A'	'CB7B'	'CB7C'	'CB7D'	'CB7E'

Tab. 6.5 – Códigos da instrução BIT.

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

INSTRUÇÃO	CÓDIGO	
SCF	'37'	(Set Carry Flag)
CCF	'3F'	(Complement Carry Flag)

Tab. 6.6 – *Set* e *Complement* do CARRY.

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: MULTIPLICAÇÃO E DIVISÃO POR POTÊNCIAS DE DOIS

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 Ø e os bits de Ø a 6 são deslocados uma posição para a esquerda (o bit 7 é copiado no CARRY). Assim, por exemplo, se o registro D contiver:

"1ØØ1ØØ1Ø" e CARRY = Ø

ao efetuarmos RLC D teremos:

D = "ØØ1ØØ1Ø1" e CARRY = 1

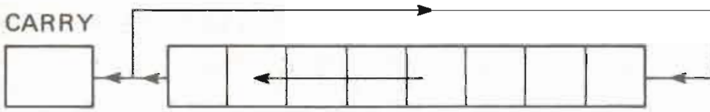


Fig. 6.3 – RLC (Rotate Left with branch Carry).

A instrução RRC é análoga à RLC, efetuando a "rotação" para a *direita* e copiando o bit Ø no CARRY. Assim, se B = "11ØØØØ11" e o CARRY = 1, ao efetuar RRC B teremos:

B = "111ØØØØ1" e CARRY = 1

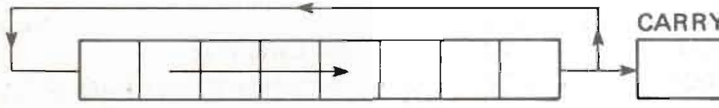


Fig. 6.4 – RRC (Rotate Right with branch Carry).

Temos também a instrução RL, que executa uma "rotação" para a *esquerda* como se o registro (ou memória) e o CARRY formassem um único registro de 9 bits; ou seja, o CARRY é introduzido no bit Ø, os bits de Ø e 6 são deslocados uma posição para a esquerda e o bit 7 é colocado no CARRY. Assim, se C = "Ø1Ø1Ø1Ø1" e CARRY = 1, ao efetuar RL C teremos:

C = "1Ø1Ø1Ø11" e CARRY = Ø

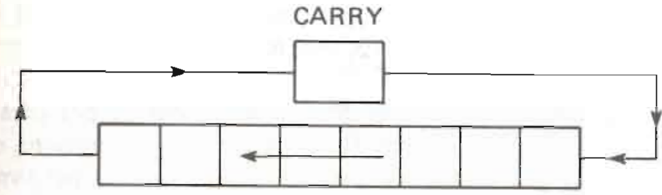


Fig. 6.5 – RL (Rotate Left through carry).

A instrução RR é análoga à RL, fazendo uma "rotação" para a *direita*. Se A = "ØØØØ1111" e CARRY = Ø, ao efetuar RR A teremos:

A = "ØØØØØ111" e CARRY = 1

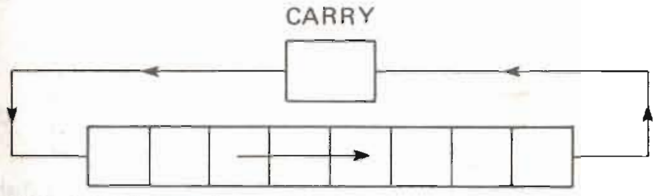


Fig. 6.6 – RR (Rotate Right through carry).

Vejamos agora a instrução SLA, que permite efetuar a multiplicação por alguma potência de 2: ela desloca todos os bits para a esquerda e "re-seta" o bit Ø (o bit 7 é copiado no CARRY). Analogamente à base 1Ø, quando multiplicamos um número por 1Ø 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 3ØØØØ:

```
3ØØØØ   LD    B,Ø    'Ø6ØØ'   ;  coloca Ø em B
3ØØØ2   SLA   C      'CB21'   ;  multiplica C por dois
3ØØØ3   RET          'C9'
```

Prog. 6.7 – Multiplicação por dois.

Ao fazer XF no HEXAMEM, o par BC é carregado com '753Ø'. Fazemos B = Ø e SLA C; e C, que era "ØØ11ØØØØ" (= 48), se torna "Ø11ØØØØØ" (= 96 = 48 x 2). Assim, obtemos 96 na tela. O fato do bit ser colocado no CARRY nos permite detectar se houve um "estouro", ou seja, resultado maior que 255.

Fig. 6.7 – SLA (Shift Left And clear least significant bit).

Por analogia, temos a instrução SRA, que desloca os bits para a direita, *sem* modificar o bit 7, e coloca o bit Ø no CARRY. Note, entretanto, que o fato do bit 7 permanecer inalterado não permite dividir números positivos maiores que 127 (= "Ø1111111") mas possibilita a divisão de números negativos (≥ –128). Entretanto, temos a instrução SRL, que desloca os 8 bits para a direita, copia o bit Ø no CARRY e "re-seta" o bit 7. Com isto, podemos dividir números positivos até 255, mas não números negativos.

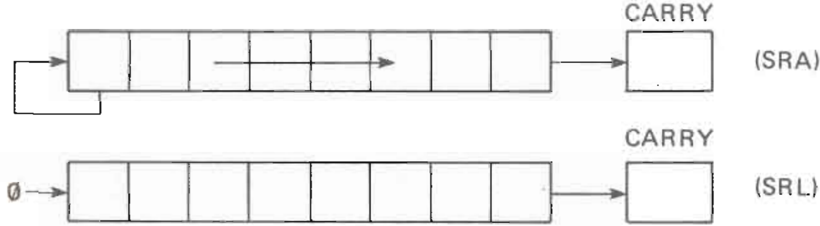


Fig. 6.8 – SRA (Shift Right And preserve most significant bit) e SRL (Shift Right and cLear most significant bit).

Exemplo

```
3ØØØØ   LD    B,Ø   'Ø6ØØ'  ; coloca Ø em B
3ØØØ2   SRL   C     'CB39'  ; divide C por dois
3ØØØ3   RET         'C9'
```

Prog. 6.8 – Divisão por dois.

Ao executar esse programa, deveremos obter 24 (48 ÷ 2).

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 com as instruções de rotação:

Instrução \ Registro	A	B	C	D	E	H	L	(HL)
RLC	'CBØ7'	'CBØØ'	'CBØ1'	'CBØ2'	'CBØ3'	'CBØ4'	'CBØ5'	'CBØ6'
RRC	'CBØF'	'CBØ8'	'CBØ9'	'CBØA'	'CBØB'	'CBØC'	'CBØD'	'CBØE'
RL	'CB17'	'CB1Ø'	'CB11'	'CB12'	'CB13'	'CB14'	'CB15'	'CB16'
RR	'CB1F'	'CB18'	'CB19'	'CB1A'	'CB1B'	'CB1C'	'CB1D'	'CB1E'
SLA	'CB27'	'CB2Ø'	'CB21'	'CB22'	'CB23'	'CB24'	'CB25'	'CB26'
SRA	'CB2F'	'CB28'	'CB29'	'CB2A'	'CB2B'	'CB2C'	'CB2D'	'CB2E'
SRL	'CB3F'	'CB38'	'CB39'	'CB3A'	'CB3B'	'CB3C'	'CB3D'	'CB3E'

Tab. 6.7 – Instruções de rotação.

Apenas a título de complementação, apresentaremos as seguintes instruções:

INSTRUÇÃO	CÓDIGO
RLCA	'Ø7'
RRCA	'ØF'
RLA	'17'
RRA	'1F'

Tab. 6.8 – Instruções de rotação com apenas 1 byte.

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 Z8Ø apenas para compatibilizá-lo com os microprocessadores 8Ø8Ø e 8Ø85 da INTEL.

Resumo

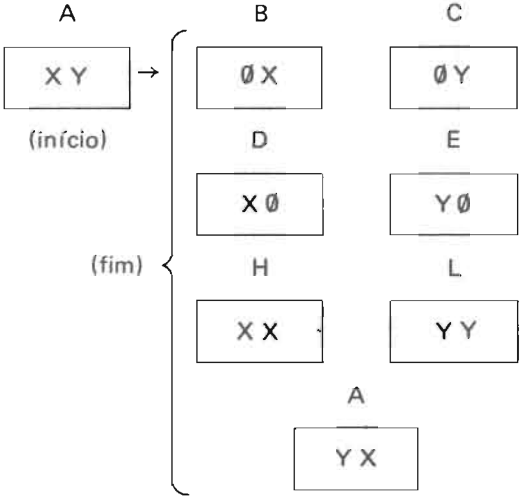
Vimos neste capítulo as operações que trabalham com bits, começando pelas operações que têm analogia com a teoria de conjuntos, ou seja, AND (intersecção), OR (união) e NOT (complemento, feita pela instrução CPL); tivemos também a operação lógica XOR, que resulta 1 apenas se os bits correspon-

dentes forem diferentes. Feito isso, vimos como "setar" e "re-setar" 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 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 um 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 π é 2Ø1/64. 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 π, deixando o resultado em BC.
4. Faça um programa que forneça como saída o número de bits em 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 1Ø 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)	mesmas observações acima
OR	(HL)	
XOR	(HL)	
AND	dado	
OR	dado	
XOR	dado	
SET	b,registro	b: Ø, 1, 2, 3, 4, 5 ou 7
RES	b,registro	
BIT	b,registro	
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		

Finalizando as Instruções em Liguagem de Máquina

Este capítulo encerra a primeira parte deste livro, dedicada a apresentar as instruções disponíveis para se trabalhar em linguagem de máquina do microprocessador Z8Ø, fazendo uma pequena revisão dos conceitos gerais e introduzindo novas instruções.

REVISÃO DE CONCEITOS – AS INSTRUÇÕES DE TROCA E OS REGISTROS IX E IY

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

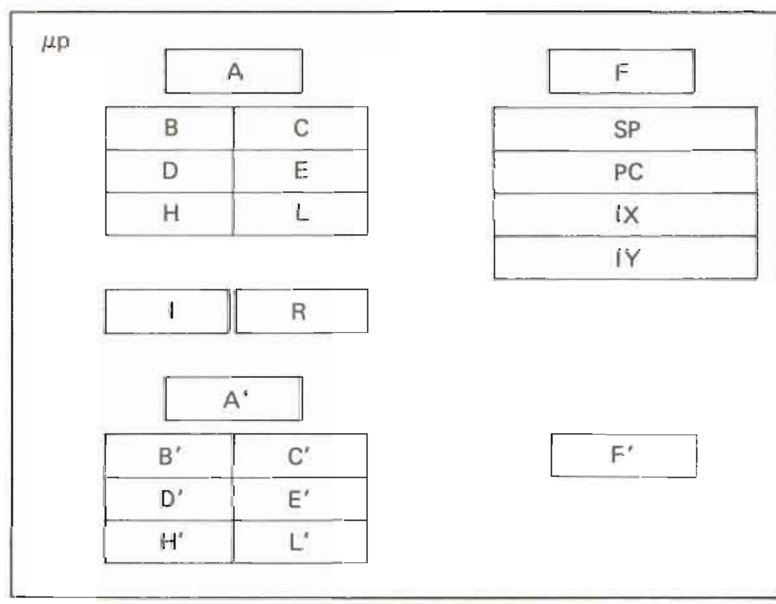


Fig. 7.1 – Como podemos imaginar os registros internos do µp.

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 o 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, não só o par BC é carregado com o argumento de USR mas também o registro PC, fazendo com que o microprocessador execute a sua sub-rotina em linguagem de máquina.

O registro F usa seus bits para dar informações com respeito a "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 PUSHs e POPs. Esse procedimento coloca os registros nas memórias da pilha indicadas por SP e, se quisermos "salvar" todos os registros, devemos fazer 4 PUSHs e 4 POPs (AF, BC, DE e HL). Entretanto, existe um método mais fácil, que economiza instruções e memória e usa a instrução:

EXX código 'D9' (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 'Ø8' (ou EX AF, AF')

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

Observação: Nunca use esta última instrução se você for operar em SLOW.

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

```
EX   DE,HL    código 'EB'
```

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

```
EX   (SP),HL   código 'E3'
```

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 endereçar 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 3ØØØØ:

```
LD   IX,3ØØØØ
```

podemos fazer:

```
LD   B,(IX + 45)
```

que coloca no registro B o conteúdo da memória 3ØØ45 (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 *'DD', se você utilizar IX*, ou *'FD', 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.

Assim, se o código de LD (HL),'24' é '3624', o código de LD (IX + '2Ø'), '24' é 'DD362Ø24' (note como o número '2Ø' foi inserido de modo a ser o terceiro byte da instrução) e o código de LD(IX),'24' é 'DD36ØØ24'. Se o código de LD HL,3ØØØØ é '213Ø75', o código de LD IY,3ØØØØ é 'FD213Ø75'. Qual seria então o código de INC IX? E de INC (IY − 2Ø)? (Utilize o apêndice 2 para ajudá-lo.)

Esses registros são aqui apresentados apenas a título de complemento, pois, apesar de serem mencionados neste livro, seu uso completo requer um estudo mais profundo da linguagem de máquina, o que você poderá ter no próximo vo-

lume desta série. O mesmo acontece com relação aos registros I e R, que não serão sequer mencionados neste livro, tendo sido colocados no desenho apenas para completar todos os registros internos disponíveis do microprocessador.

O registro IY tem, entretanto, uma peculiaridade interessante: ao chamar uma sub-rotina em linguagem de máquina, o programa interpretador do TK coloca automaticamente em IY o valor '4ØØØ' (16384), que corresponde exatamente ao endereço a partir de onde, na RAM, estão colocadas as variáveis do programa interpretador. Você lembra da variável D-FILE usada no ANTISCROLL? Seus endereços eram 16396 e 16397. Assim, para obter D-FILE em uma sub-rotina em linguagem de máquina e colocá-la, por exemplo, no par HL, bastaria fazer:

```
LD   L,(IY + 12)
LD   H,(IY + 13)
```

desde que você não altere o valor de IY, naturalmente. Ao voltar ao BASIC, o registro IY é novamente carregado com '4ØØØ' pelo programa interpretador.

Os registros IX e IY podem ser usados *praticamente* em *todas* as instruções que utilizam o par HL. Há poucas exceções e, em caso de dúvida, basta consultar o apêndice 2.

Observação: Nunca tente utilizar o registro IX se você operar em SLOW.

COMPARAÇÃO DE BLOCOS E INSTRUÇÕES FINAIS

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 e 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:

INSTRUÇÃO	CÓDIGO
CPI	'EDA1' (compara A com memória apontada por HL, INC HL e DEC BC)
CPD	'EDA9' (idem, mas DEC HL)
CPIR	'EDB1' (idem a CPI mas repete até BC ser zero)
CPDR	'EDB9' (idem a CPD mas repete até BC ser zero)

Tab. 7.1 – Comparação de blocos.

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 30000 e 30099 existe o número 117, basta fazer:

```
LD    BC,100
LD    HL,30000
LD    A,117
CPIR
```

Prog. 7.1 – Uso de CPIR.

Feito isto, basta verificar qual o conteúdo de HL para saber a posição em que está o número 117. Obviamente, se houver mais que uma memória com 117 nessa região, só a primeira será detectada; se não houver nenhum 117, 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 Z80. Entretanto, salientamos que muitas parecerão confusas, mas não se preocupe com isto pois seu uso completo será explicado em detalhes no próximo volume e, se você se interessar por linguagem de máquina, poderá então entendê-las completamente. Elas serão explicadas resumidamente caso você esteja curioso e queira fazer algumas experiências; não serão vistas, entretanto, as instruções que utilizam os registros I e R.

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 – ver apêndice 2.)
- 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:

```
PUSH   AF     ; coloca AF no topo da pilha
POP    BC     ; coloca o conteúdo do topo da pilha em BC
BIT    4,C    ; testa o bit 4 do registro C (ver figura 7.1)
(Tente explicar por que!)
```

Prog. 7.2 – Teste da flag AC.

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

Assim, o registro F, completo fica:

7	6	5	4	3	2	1	0
S	Z		AC		P/O	N	C

Fig. 7.1 – Flags no registro F.

Note que os bits 5 e 3 *não* são utilizados devido à compatibilidade com os microprocessadores 8080 e 8085.

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

NOP código *'00'*

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. À 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 NOPs em seguida. Naquela região é possível, mais tarde, substituir os NOPs por outras instruções, sem problema; mas, se não houverem os NOPs e você quiser acrescentar instruções no meio de um programa, então

terá de *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 NOPs quantos forem os bytes ocupados pela instrução, obviamente colocados a partir do endereço inicial da instrução. Suponha que você tivesse:

```
30000  LD    A,32
30002  EX    DE,HL
30003  LD    (IX + 33),160    'DD3621A0'
30007  LD    IY,30000         'FD213075'
30011  RET
```

Prog. 7.3 – Programa sem sentido.

Para retirar a instrução LD (IX + 33), 160, por exemplo, basta fazer:

```
POKE  30003,0
POKE  30004,0
POKE  30005,0
POKE  30006,0
```

(Note que o código de NOP é '00'.)

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 ('10'), 24 ('18'), 32 ('20'), 40 ('28'), 48 ('30') e 56 ('38'). Assim, temos:

INSTRUÇÃO	CÓDIGO
RST 0	'C7' (equivalente a CALL 0)
RST 8	'CF' (equivalente a CALL 8)
RST 16	'D7' (equivalente a CALL 16 ou CALL '10')
RST 24	'DF' (equivalente a CALL 24 ou CALL '18')
RST 32	'E7' (equivalente a CALL 32 ou CALL '20')
RST 40	'EF' (equivalente a CALL 40 ou CALL '28')
RST 48	'F7' (equivalente a CALL 48 ou CALL '30')
RST 56	'FF' (equivalente a CALL 56 ou CALL '38')

Tab. 7.2 – A instrução RST.

No TK, fazer RST 0 produz o mesmo efeito que NEW mas, além disso, coloca RAMTOP no seu valor máximo, coisa que a instrução NEW não faz (assim seu efeito é como se desligássemos o computador).

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 programa interpretador, que poderão eventualmente ser utilizadas em outros programas. Em capítulos futuros daremos alguns exemplos do uso de RST.

Quando o microprocessador tiver de trabalhar com dispositivos externos (impressora, gravador, TV etc.) poderá ocorrer de ele ser interrompido por esses mesmos dispositivos quando da execução de algumas funções, abandonando momentaneamente à execução do programa principal. Existem então duas instruções que servem para dizer se ele pode ou não ser interrompido (por exemplo, se ele estiver executando uma parte importante do programa, não é conveniente interrompê-lo). São elas:

```
EI   código  'FB'  (Enable Interrupt)   (permite a interrupção)
DI   código  'F3'  (Desable Interrupt)  (não permite a interrupção)
```

Ao ser interrompido, se houver permissão, o microprocessador passará a executar uma sub-rotina que deve terminar com uma das seguintes instruções:

```
RETI   código   'ED4D'  (RETurn from Interrupt)
```

ou:

```
RETN   código   'ED45'  (RETurn from Non-maskable interrupt)
```

Além disso, o microprocessador tem vários modos diferentes de interrupção, selecionáveis pela instrução IM (Interrupt Mode):

```
IM 0   código   'ED46'
IM 1   código   'ED56'
IM 2   código   'ED5E'
```

Não fique desesperado, isso não nos adiantará muito, por enquanto!

Um exemplo de uso de interrupção é o funcionamento da impressora: o microprocessador envia um caractere (ou uma linha de pontos) a ser impresso para a impressora e, antes de mandar o próximo, espera ser *interrompido* pela impressora, que o avisará que terminou de escrever o caractere e está aguardando o próximo. Você pode então notar que devem existir instruções para que o microprocessador *envie* dados para fora e *receba* dados de dispositivos que não fazem parte da memória. Essas instruções são: OUT, que envia para dado dispositivo o conteúdo dos registros A, B, C, D, E, H ou L, e IN, que coloca o dado enviado pelo dispositivo em qualquer um desses registros. Analogamente a LDI, LDR, LDIR e LDDR, temos as instruções INI, IND, INIR, INDR, OUTI, OUTD,

OTIR e OTDR para enviar (ou receber) blocos de dados de dispositivos externos.

Finalmente, temos três instruções que facilitam o uso de números *decimais* (com certas restrições). São elas:

DAA (Decimal Adjust Accumulator)
RLD (Rotate Left Decimal)
RRD (Rotate Right Decimal)

Novamente frisamos que você não deve se preocupar com essas instruções; elas foram aqui apresentadas apenas a título de complemento e serão detalhadas no próximo volume.

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 exemplo:

```
LOOP  ADD   A,C
      INC   C
      LD    (HL),A
      DEC   B
      JR NZ, LOOP
      CALL  NADA
      JP    LOOP
NADA  NOP
      NOP
      RET
```

Prog. 7.4 — Uso de LABELS.

Neste exemplo, LOOP e NADA são LABELS. Suponha que agora você queira colocar os códigos na memória e que o programa comece no endereço 30000. Os LABELS que estão à esquerda serão então calculados:

LOOP = 30000
NADA = 30012

(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 de sub-rotinas, os LABELS podem ser substituídos diretamente:

```
CALL NADA  –  (CALL 30012  código 'CD3C75')  30012 = '753C'
JP   LOOP  –  (JP   30000  código 'C33075')  30000 = '7530'
```

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

```
JR NZ LOOP  –  (JR NZ -6  código '20FA')  -6 = 'FA'
```

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, além de transformar automaticamente os mnemônicos em códigos, calculam os endereços relativos aos LABELS.

Resumo

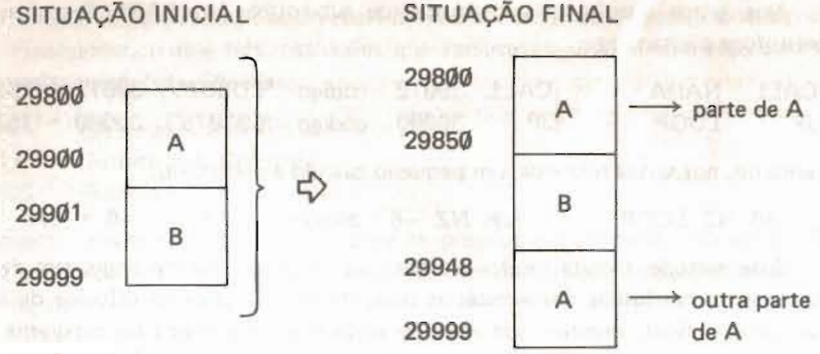
Fizemos uma rápida revisão dos conceitos vistos até este capítulo e introduzimos todos os registros internos do microprocessador, incluindo A', B', C', D', E', H', L', I, R, IX e IY.

Terminamos de apresentar as instruções em linguagem de máquina (embora algumas não sejam muito utilizadas neste volume), 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 de LABELS, que facilita a escrita e posterior leitura do programa. Vamos iniciar agora a aplicação mais imediata das instruções em linguagem de máquina para o TK. Começaremos por explicar alguns conceitos da estrutura interna do computador, como a organização da memória e métodos de comunicação com a tela e o teclado, acompanhados de exemplos bastante interessantes.

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 dada região de memória. Para verificar o funcionamento do programa, transfira um bloco B (29901 a 29999) para dentro de um bloco A (29800 a 29900), inserindo-o a partir de 29850.



Utilize LABELS para facilitar a escrita do programa.

Sugestão: Faça RAMTOP igual a 298ØØ e escreva seu programa a partir da memória 3ØØØØ. Para verificar se seu programa funcionou, utilize um programa em BASIC que faça um PEEK das regiões utilizadas *antes* e *depois* de executar o programa em linguagem de máquina.

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 NOPs. 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 caso 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 utilizando POKE (como fizemos nos exercícios dos capítulos 3 e 5).

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 (após ter feito o exercício, verifique sua resposta no apêndice 2 e procure escrever também os códigos hexadecimais para cada instrução).

```
LD D,(BC)
LD A,(IX + 16)
ADD HL,BC
EXX BC,DE
SUB HL,DE
LD  (15Ø95),A
LD  (37Ø4Ø),22
AND (IX + '2Ø')
OR  IY + 16
XOR HL
INC IX
LD B,(HL)
LD BC,(3ØØØØ)
ADC HL,SP
ADD BC,SP
LD  BC,DE
LD  HL,SP
EXX SP,(HL)
CALL NC, 31ØØØ
RST 16
LDDR
BIT 1,(IX + 1Ø)
SCF
NEG B
LD DE,1434Ø
LD DE,(1434Ø)
RL  (HL)
RL  HL
RR  H
RES Ø,(IY)
RES Ø,IY
JR M, 12Ø
JR C, 2Ø7
JP C, 2Ø7
PUSH A'F'
POP BC
POP CD
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
```

5. Utilizando a instrução EX (SP),HL, faça um programa que "salve" nas memórias 16507 e 16508 (mencionadas no capítulo 5) o endereço de retorno de uma sub-rotina e não o valor de SP.

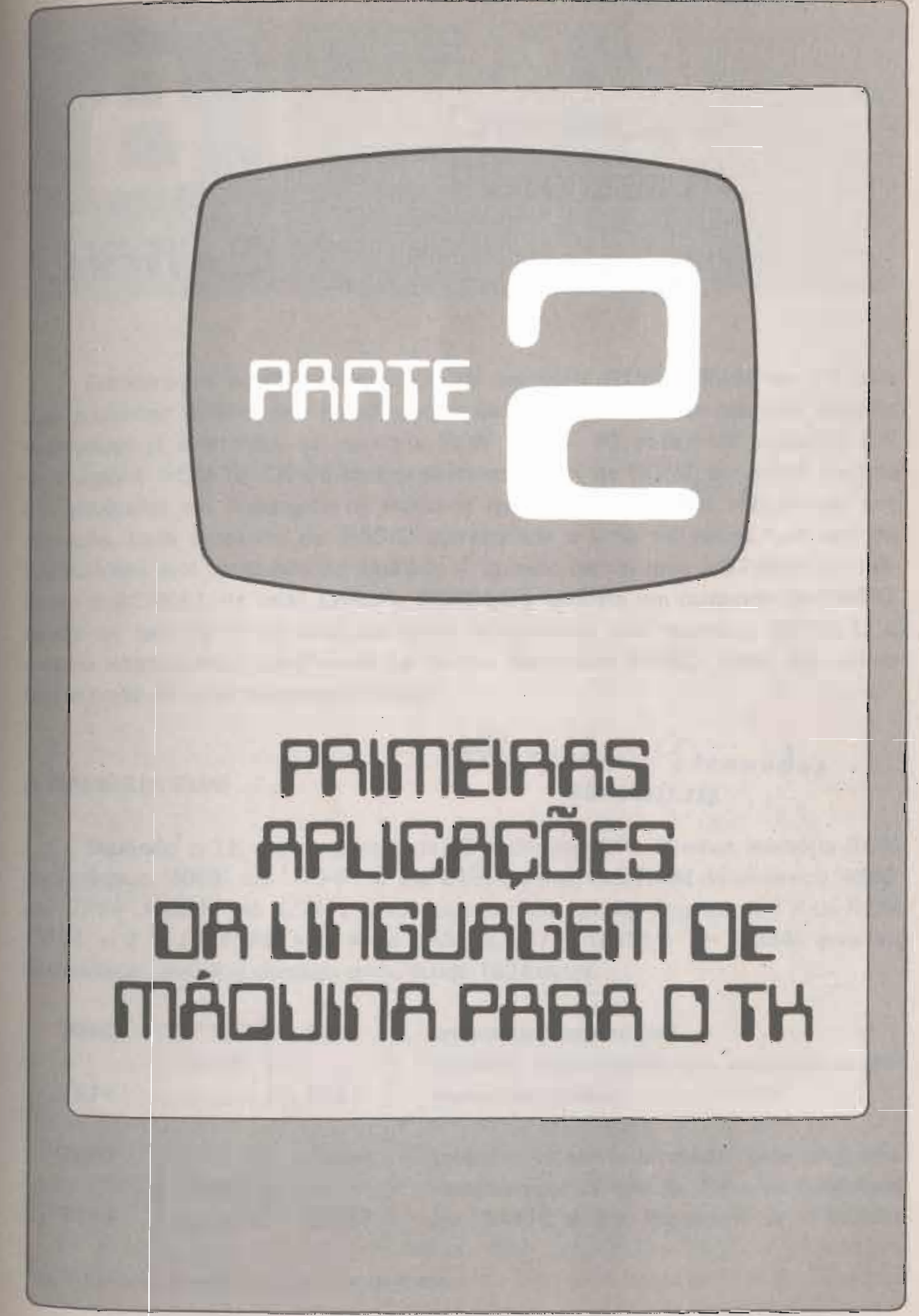
Instruções vistas neste capítulo

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

Instruções usando { IX } – prefixo 'DD' / { IY } – prefixo 'FD'

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 }



Organização de Memória do TK

Estudaremos agora a organização da memória ROM e RAM do TK para que possamos utilizar nos nossos programas em linguagem de máquina algumas sub-rotinas já existentes na memória ROM. O TK 82 possui os primeiros 8 K de memória ROM (o TK 85 tem os primeiros 10 K de ROM), onde está contido um programa em linguagem de máquina que consta de várias sub-rotinas; por exemplo, cada comando do BASIC corresponde a uma sub-rotina que está na ROM. Você está lembrado do capítulo 4 quando demos uma sub-rotina que realizava o SCROLL da tela? Como o SCROLL é também um comando do BASIC, existe de fato na ROM uma sub-rotina semelhante, que realiza o SCROLL, o mesmo acontecendo para todos os outros comandos BASIC. Essas sub-rotinas fazem parte do programa interpretador.

A MEMÓRIA ROM

Supondo o TK com expansão de 16 K de memória, teremos memória ROM do endereço '0000' até '1FFF' (0 até 8191), e memória RAM do endereço '4000' até '7FFF' (16384 até 32767). Você pode verificar que de fato temos 8 K de ROM (8191 − 0 + 1 = 8192) e 16 K de RAM (32767 − 16384 + 1 = 16384), pois em computação cada K é contado como sendo 1024 bytes.

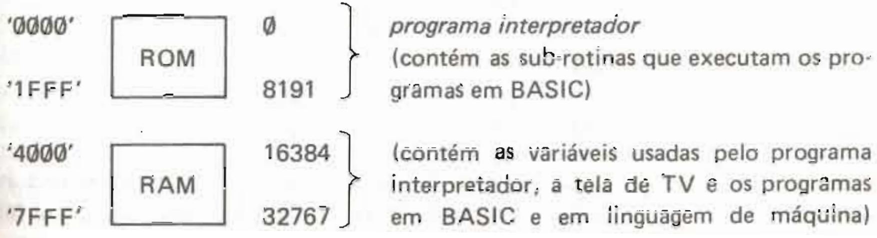


Fig. 8.1 — Memória do TK com 16 K de RAM.

Você deve lembrar que no início do livro verificamos o valor de RAMTOP com 16 K de memória RAM, que de fato era 32768, ou seja, o primeiro byte após o *último* byte da RAM (32768 = '8000'). Vimos também que a memória *poderia* ir de 0 até 65535 ('0000' a 'FFFF') e que, portanto, no esquema da figura 8.1 existem alguns espaços que poderiam ser preenchidos com memória RAM, e é o que de fato acontece quando você coloca a expansão de 64 K num TK 82.

Apresentaremos então o endereço em hexadecimal de algumas sub-rotinas que estão na ROM. Note, entretanto, que um estudo completo do programa da ROM foge dos objetivos deste volume e, portanto, ele não será visto aqui minuciosamente.

'0808' . . . . . rotina para colocar um caractere na tela
'03C3' . . . . . NEW
'0730' . . . . . LIST
'0A2A' . . . . . CLS
'0EAF' . . . . . RUN

Assim, por exemplo, usando HEXAMEM, coloque a partir da memória 30000 o seguinte:

```
30000  CALL  RUN  'CDAF0E'
30003  RET        'C9'
```

Prog. 8.1 – Utilização da sub-rotina de RUN na ROM.

Após colocar *P*, o programa pára. Reinicie o programa mas, em vez de fazer RUN, faça RAND USR 30000 (comando direto) e NEW LINE e... o programa começará novamente! Coloque os comandos LIST, CLS e NEW nas memórias seguintes; faça memória inicial = 30004 e coloque:

```
30004  CALL  LIST  'CD3007'
30007  RET         'C9'
30008  CALL  NEW   'CDC303'
30011  RET         'C9'
30012  CALL  CLS   'CD2A0A'
30015  RET         'C9'
```

Em outras palavras, cada chamada de sub-rotina executará as ordens existentes na ROM correspondentes a cada comando. Para exemplificar, digite NEW e execute o seguinte programa auto-explicativo (esperamos que você tenha se lembrado de reservar a memória...):


```
  10 SLOW
  20 FOR I=1 TO 22
  30 PRINT "AO TERMINAR A TELA V
OU ME APAGAR"
  40 NEXT I
  50 RAND USR 30012
  60 PRINT "AGORA VOU ME LISTAR"
  70 GOSUB 200
  80 CLS
  90 RAND USR 30004
 100 GOSUB 200
 110 CLS
 120 PRINT "E AGORA VOU ME AUTOD
ESTRUIR","ADEUS...SNIFF...SNIFF.
.."
 130 GOSUB 200
 140 RAND USR 30008
 200 FOR T=0 TO 40
 210 NEXT T
 220 RETURN
```

Prog. 8.2 – Uso de várias sub-rotinas da ROM.

A rotina que coloca um caractere na tela (endereço '0808') será utilizada no próximo capítulo. No capítulo 10 iremos usá-la juntamente com uma explicação interessante sobre as tabelas existentes na ROM, onde estão os *moldes* para a construção dos caracteres. No próximo volume, estudaremos com detalhes a estrutura da ROM do TK, analisando as sub-rotinas correspondentes a todos os comandos do BASIC, além de estudar *como* o programa interpretador controla e organiza todo o sistema.

A MEMÓRIA RAM

Como sabemos, o programa interpretador, *não* podendo armazenar suas variáveis na memória ROM, armazena-as no começo da RAM, mais precisamente nas memórias entre 16384 (inclusive) e 16508. Algumas dessas variáveis já foram vistas em capítulos anteriores, como é o caso de RAMTOP (16388 e 16389), D-FILE (16396 e 16397), S-TOP (16419 e 16420) e o número da linha para a posição PRINT (16442) (note que todos os endereços estão entre 16384 e 16508. No manual do TK você encontrará a lista das principais variáveis usadas pelo programa interpretador. Aprenderemos agora mais algumas variáveis e, para entendermos seu uso, faremos um esquema mais detalhado da memória RAM:

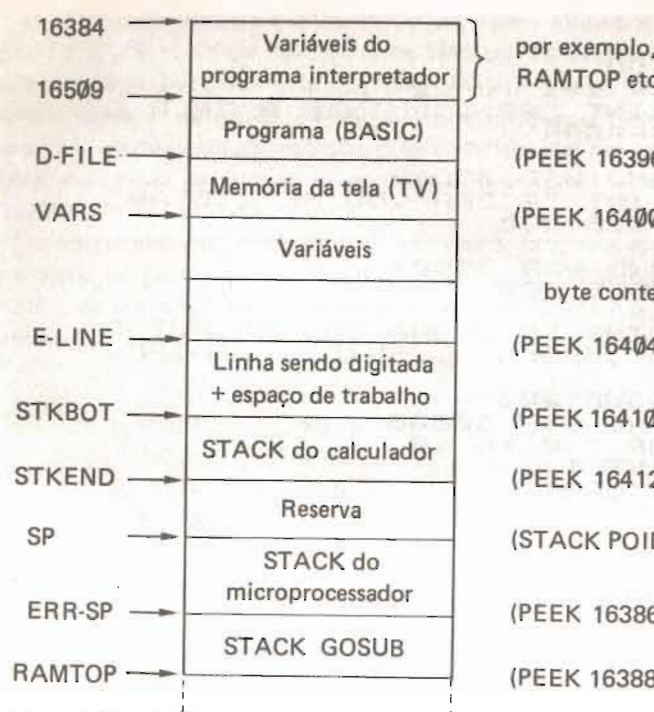


Fig. 8.2 – Memória RAM.

Como podemos ver, as variáveis do programa interpretador estão entre as memórias 16384 (inclusive) e 16508; e, entre essas variáveis, estão D-FILE (16396 e 16397), VARS (16400 e 16401), E-LINE (16404 e 16405), STKBOT (16410 e 16411), STKEND (16412 e 16413), ERR-SP (16386 e 16387) e RAMTOP, que servem para indicar as "fronteiras" existentes na memória RAM. O programa em BASIC começa *sempre* na memória 16509, mas todas as demais regiões têm seus endereços iniciais alteráveis, sendo calculados pelo programa interpretador e colocados no início da memória RAM, nas respectivas variáveis. Assim, se você, por exemplo, acrescentar uma linha ao programa em BASIC, todos os endereços das fronteiras serão movidos "para baixo", deslocando-se, obviamente, o conteúdo de cada região. Logo depois do programa em BASIC fica o conteúdo da tela de TV, colocado a partir de um endereço dado por D-FILE (que está no início da memória RAM, nos endereços 16396 e 16397); assim, a primeira posição da tela (ver capítulo 4) pode ser achada por PRINT PEEK 16396 + 256* PEEK 16397 + 1.

A variável E-LINE já foi mencionada no capítulo 3; como os programas em BASIC começam *sempre* na memória 16509, você pode agora perceber o porquê

da subtração feita no capítulo 3 (E-LINE – 16509) para calcular o tamanho ocupado pelo programa, tela de TV e variáveis.

Cada instrução do programa em BASIC ocupa 2 bytes para armazenar o número da linha, 2 bytes para colocar o tamanho da instrução, *n* bytes para a instrução em si e, no fim, um NEW LINE.

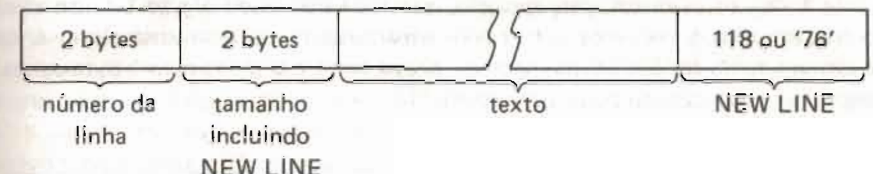


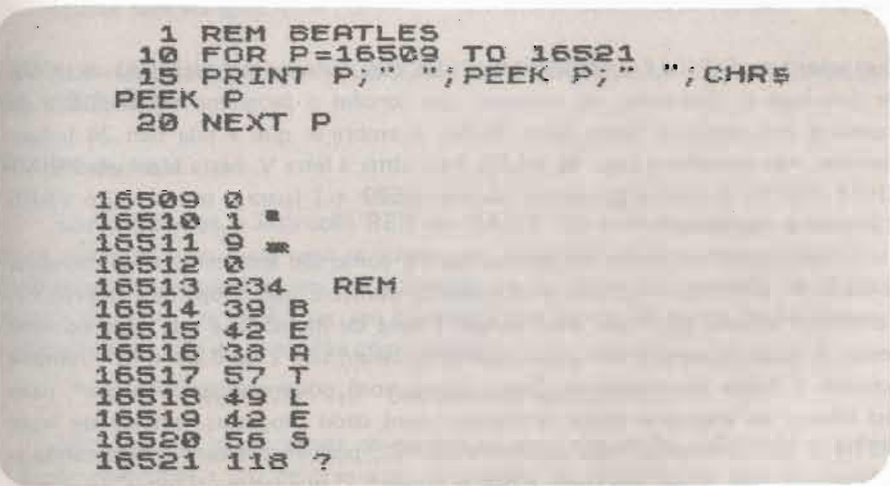
Fig. 8.3 – Instrução na memória.

Assim, se sua primeira linha no programa for:

1 REM BEATLES

o código de REM estará na memória 16513 (16509 + 4), pois o número da linha está nas memórias 16509 e 16510, o tamanho da linha nas memórias 16511 e 16512, a letra B está na memória 16514 e assim por diante, sendo que na memória 16521 está um código de NEW LINE. Qual seria, nesse caso, o valor de D-FILE?

Experimente executar este programa:



```
   1 REM BEATLES
  10 FOR P=16509 TO 16521
  15 PRINT P;" ";PEEK P;" ";CHR$
PEEK P
  20 NEXT P


16509 0
16510 1 ■
16511 9 ▀
16512 0
16513 234    REM
16514 39 B
16515 42 E
16516 38 A
16517 57 T
16518 49 L
16519 42 E
16520 56 S
16521 118 ?
```

Prog. 8.3 – Visualização do conteúdo da linha REM.

Você saberia explicá-lo?

Após a tela de TV, ficam as variáveis utilizadas pelo programa em BASIC. O endereço inicial dessa região é dado por VARS (16400 e 16401). Note que as variáveis numéricas são armazenadas de uma forma especial para permitir que *qualquer* número que esteja dentro da precisão do computador possa ocupar *sempre* 5 bytes. Isto é feito pelo programa interpretador, que transforma internamente todos os números para notação científica utilizando alguns bits para expoente, um para o sinal e os outros para a mantissa; assim, uma variável numérica de um programa em BASIC na memória ocupa sempre 5 bytes mais 1 byte correspondente a cada letra do nome da variável.

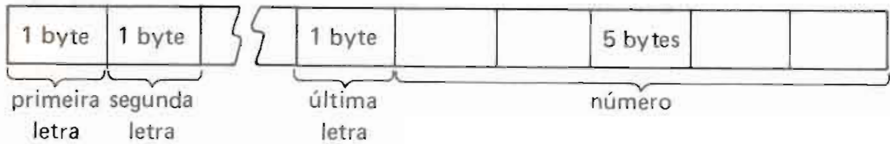


Fig. 8.4 – Variáveis numéricas na memória.

Observe o seguinte programa:

```
10  LET LOVE = 20.5
20  STOP
```

Prog. 8.4 – Exemplo de colocação de variáveis.

Que valor tem D-FILE? E VARS? Use PEEK para obter a letra *V* da palavra LOVE de dois lugares diferentes: da memória que contém o programa em BASIC e da memória das variáveis (após fazer RUN). (Lembre-se que a tela tem 24 linhas. Por isso, não estranhe o valor de VARS. Para obter a letra V, basta fazer um PRINT CHR$ (PEEK(n)) com argumentos valendo 16509 + 7 (para o programa) e VARS + 2 (para as variáveis)).

Você pode encontrar no manual do TK como são armazenadas na memória as variáveis STRING, matrizes e variáveis de controle para loops de FOR-NEXT. Lembre-se sempre que cada *letra* ocupa 1 byte de memória e que cada *número* ocupa 5 (cada número e *não* cada algarismo; assim, seja 2 ou $3{,}74 \times 10^{14}$, ambos ocupam 5 bytes de memória). Dessa forma, você pode usar um "truque", caso seu espaço de memória esteja acabando: num dado programa, em vez de fazer GOTO 2 (por exemplo), faça GOTO VAL "2", pois você estará armazenando o caractere 2, que ocupa um byte, e não o número 2, que ocupa 5 bytes. (Isso vale para todas as instruções que usam números!) Caso você esteja trabalhando com matrizes, perceba a redução de memória: ao trabalhar com uma matriz de 100 números com 2 algarismos cada, se ela for numérica você precisará de 500 bytes

para guardá-los mas, se você os transformar em STRINGs necessitará apenas de 200 bytes para armazená-los!

Bem, após as variáveis, está um byte de fronteira contendo sempre o valor '80' (figura 8.2); a seguir, vem uma região onde fica a linha que está sendo digitada. O endereço inicial dessa região é dado por E-LINE. Temos então uma região usada pelo STACK do calculador que faz as contas em BASIC; essa região é essencial para que possam ser feitas as contas e não entraremos em detalhes sobre seu funcionamento. Esse STACK nada tem a ver com o STACK POINTER do microprocessador; de fato, o STACK POINTER é também indicado no esquema por SP e tem o seu "campo de trabalho" limitado entre STKEND e ERR-SP (mencionado no capítulo 5), enquanto o STACK do calculador fica entre STKBOT e STKEND. Como em BASIC também podemos ter sub-rotinas, é necessário usar o mesmo "truque" do STACK POINTER do microprocessador para guardar os endereços de retorno; eles são colocados no final da memória RAM entre ERR-SP e RAMTOP. Você percebe o que acontece quando altera o valor de RAMTOP? O TK "pensa" que tem sua memória reduzida e procura "espremer" toda a memória RAM, assim ele nunca incomodará o que fica após RAMTOP!

Você deve estar lembrado de quando falamos que, ao reduzir RAMTOP para colocar lá o programa em linguagem de máquina, esse não era afetado pelo NEW mas também não poderia ser gravado em fita usando SAVE, pois o computador armazena em fita apenas as partes da memória RAM correspondentes às variáveis do programa interpretador, ao programa em BASIC e suas variáveis e à tela. Vamos ver então outros lugares para colocar nossas sub-rotinas em linguagem de máquina, sem ser após RAMTOP.

OUTROS LUGARES PARA COLOCAR OS PROGRAMAS EM LINGUAGEM DE MÁQUINA

A Instrução REM

Como sabemos, a instrução REM em BASIC não executa nenhum comando; assim, ela é um bom lugar para se colocar um programa em linguagem de máquina se desejamos armazená-lo em fita (cuidado, ele também será apagado ao digitar NEW!). Portanto, se você tiver um programa que ocupe 60 bytes, basta colocar a primeira instrução do programa como sendo:

1 REM ???? ::: 60 caracteres quaisquer....????

Não importa quais sejam os caracteres pois eles serão substituídos pelos que correspondem a seu programa em linguagem de máquina; no entanto, é conveniente usar os números de 0 a 9 em seqüência, várias vezes, para facilitar a contagem:

1 REM 01234567890123...

Assim, se quisermos colocar numa linha REM o seguinte programa:

```
LD   B,'FF'   'Ø6FF'
LD   C,B      '48'
RET           'C9'
```

Prog. 8.5 – Programa que será colocado numa linha REM.

teremos de fazer, primeiramente,

```
1 REM Ø123   (pois o programa tem 4 bytes)
```

para reservar espaço na memória.

Ora, qual o endereço do número Ø na linha de REM? Volte algumas páginas e você verá que esse número está na memória 16514, o número 1 na memória 16515 e assim por diante...

Faça então:

```
POKE   16514,6      (Ø6 ⇨ LD B,)
POKE   16515,255    (255 = 'FF')
POKE   16516,72     (72 = '48')    (= LD C,B)
POKE   16517,2Ø1    (2Ø1 = 'C9')   (= RET)
LIST
```

O que aconteceu com a instrução REM? Basta olhar para o apêndice 2 e verificar que o caractere correspondente a 'Ø6' é ▞ , a 'FF' é COPY, a '48' não existe (aparecendo, portanto, um ponto de interrogação (?)) e a 'C9' é TAN.

Obviamente você poderia construir esse REM diretamente mas teria de reservar espaço para o terceiro caractere, pois ele não tem tecla correspondente, e, a seguir, fazer POKE 16516,72; além do mais, para conseguir o COPY, sendo este uma *key-word*, você deveria fazer THEN COPY e, a seguir, retirar o THEN.

1º passo: 1 REM ▞ THEN COPY [L]

2º passo: 1 REM ▞ COPY [L]

3º passo: 1 REM ▞ COPY.TAN [L] (o ponto serve para reservar lugar)

A seguir, basta "mandar a linha para cima" e fazer POKE 16516,72.

Vamos executar o programa? Ora, basta fazer:

```
PRINT USR 16514
```

e, como esperado, iremos obter 65535 ('FFFF').

Naturalmente, se o programa for longo, esses métodos não são aconselháveis e você pode usar o HEXAMEM, indicando como endereço inicial a memória 16514 (não se esqueça de reservar as memórias com o REM). Obviamente, nesse caso, você pode retirar da linha 3Ø do HEXAMEM a mensagem que diz que a memória deve ser maior ou igual a 3ØØØØ.

Antes de prosseguirmos, são necessários alguns comentários com relação à estrutura da instrução REM; existem dois números que podem causar algumas "dores de cabeça" se tiverem de ser introduzidos numa instrução REM, que são '76' e '7E'. O primeiro deles não causa conseqüências desastrosas, apenas confundirá um pouco a listagem do seu programa, pois '76' é o código de NEW LINE (note, entretanto, que esse é também o código de HALT. Assim, o único meio de aparecer '76' numa instrução de REM é como sendo um dado ou *parte de um endereço*). Experimente colocar no computador uma instrução REM, por exemplo:

```
1 REM Ø1234567890123456789
```

e faça:

```
POKE   16519,118 (118 = '76')
LIST
```

Observe que apareceu uma linha (8739) que na realidade *não* existe. Ela faz parte da linha 1. Verifique isto movendo o cursor [>] (se houver outras linhas de programa). Isto acontece pois você está colocando um código de NEW LINE, fazendo com que o computador "pense" que terminou a linha 1 e causando uma "divisão" da linha na tela. Mas, como acabamos de ver, no começo da linha na memória do programa os primeiros dois bytes armazenam o número da linha e os próximos dois o comprimento da mesma; assim, apesar de parecerem duas linhas na listagem, serão consideradas uma só *pelo programa*. Note que o número 118 ('76') fica invisível na listagem da tela...

Entretanto, há um pequeno problema: se você quiser alterar a "segunda parte" da linha, a solução não será trivial. Tente colocar o cursor na linha 1 e trazê-la para a região de edição usando EDIT... a segunda parte não descerá. Portanto, antes de fazer isso, faça um POKE anulando temporariamente o NEW LINE:

```
POKE   16519,6   (poderia ser qualquer coisa menos '76' ou '7E')
LIST
```

aparecerá o caractere ▞ no lugar de NEW LINE. Agora você pode trazer a linha "para baixo", efetuar as alterações necessárias e, após colocá-la novamente na parte superior da tela, refazer:

```
POKE   16519,118
```

Com relação ao '7E', muito cuidado deve ser tomado. Suponha que você reservou um espaço na sua instrução REM e, após colocar o programa, notou que devia acrescentar mais alguns bytes ou, eventualmente, tirar alguns bytes do final porque eles sobraram. Se o número '7E' não estiver na instrução REM, você pode utilizar EDIT e colocar ou retirar quantos bytes você quiser; mas, se o número '7E' estiver na instrução, *não* use EDIT. Na listagem, o número '7E' é *invisível* e os 5 bytes que o seguem também o são, apesar de estarem na memória. Se você usar EDIT esses 6 bytes *sumirão* e você os perderá para sempre... De fato, faça:

```
1 REM  Ø123456789
```

e, a seguir:

```
POKE  16514,126    (126 = '7E')
LIST
```

Veja: os números Ø, 1, 2, 3, 4 e 5 estão invisíveis... Feito isto, vamos fazê-los reaparecer:

```
POKE  16514,6
LIST
```

(Você pode explicar o caractere que aparece em seguida ao REM?)

Vamos então fazer novamente:

```
POKE  16514,126
```

e, em seguida, um EDIT para trazer a linha para baixo. Coloque a linha novamente no programa (basta apertar NEW LINE) e, em seguida, faça outra vez:

```
POKE  16514,6
LIST
```

Como esperado, os 6 bytes desapareceram...

Isso acontece porque o programa interpretador utiliza este número ('7E') para tornar as listagens "apresentáveis" quando aparecem números no programa em BASIC. Não nos interessam muito os detalhes do porquê, pois isto implicaria um estudo mais profundo da ROM. O que importa é *como* fazer para aumentar a REM, se necessário. A idéia seria colocar uma linha 2 com outro REM e tentar juntá-las, mas há um outro probleminha: como vimos anteriormente, cada linha de programa usa os primeiros 2 bytes para colocar o número da linha e, em seguida, mais 2 bytes invisíveis para a listagem onde está armazenado o comprimento da linha. Assim, se você tentar retirar o NEW LINE da primeira linha para tentar juntá-las, o programa interpretador continuará tentando interpretar a linha 2 do mesmo ponto devido aos 2 bytes da linha 1 que dizem seu

comprimento. Assim, devemos alterar também esses 2 bytes. Um bom método para aumentar a linha 1, se ela contiver o byte '7E', consiste no seguinte:

a) introduza uma linha 2 REM mais um número de caracteres suficientes para aumentar seu programa,
b) coloque o seguinte programa que fará os cálculos necessários (não colocaremos os números das linhas pois você deve colocá-los onde for mais conveniente de maneira a não estragar algum eventual programa que já esteja na memória):

```
LET   C = PEEK  16511 + 256 * PEEK 16512
LET   X = 16515 + C
LET   X = X + PEEK X + 256 * PEEK (X + 1) - 16511
POKE  16511,X - 256 * INT (X/256)
POKE  16512,INT (X/256)
STOP
```

Prog. 8.6 – Programa para aumentar uma REM que contém o byte '7E'.

Perceba: em C, colocamos o comprimento da primeira linha, a seguir, em X, colocamos o *endereço* em que está o comprimento da segunda linha. Com isto, podemos então calcular o comprimento da segunda linha, somá-lo a X e subtrair 16511 para obter o comprimento total das 2 linhas; a seguir, modificamos os bytes da linha 1 para que eles contenham agora o comprimento total.

Após executar esse programa, você poderá retirá-lo, se quiser. Agora a linha 2 faz parte da linha 1, apesar de na listagem elas parecerem ainda separadas devido ao fato de o NEW LINE final da linha 1 ainda estar lá. Você pode verificar isso tentando mover o cursor (*mas não edite a linha!* Lembre-se do '7E'!). Se você quiser retirar este NEW LINE, ou melhor, substituí-lo por qualquer outro caractere que torne as coisas mais visíveis, basta fazer:

```
POKE  (C + 16512),6  ——→ (ou qualquer outro caractere visível)
          ↓
   comprimento da antiga
          linha 1
```

ou então acrescente a linha acima antes do STOP.

Se você quiser diminuir o tamanho da linha REM, deve fazer o seguinte (esse programa reduz no mínimo 6 bytes; você saberia explicar por quê?):

```
LET   X = endereço do último byte que você quer preservar
LET   Y = X - 16511
LET   Z = PEEK 16511 + 256 * PEEK 16512 - Y - 4
POKE  16511,Y - 256 * INT (Y/256)
POKE  16512,INT (Y/256)
POKE  X + 1,118
POKE  X + 2,0
POKE  X + 3,2
POKE  X + 4,Z - 256 * INT (Z/256)
POKE  X + 5,INT (Z/256)
STOP
```

Prog. 8.7 – Programa para diminuir uma REM com o byte '7E'.

As primeiras duas linhas calculam o novo comprimento que é colocado em Y, e, em Z, colocamos o tamanho de uma nova linha 2 que será "criada"; a seguir, colocamos o novo comprimento nos respectivos bytes e "criamos" a linha 2, colocando na memória o número da linha e o tamanho. Ao executar o programa, aparecerá então uma nova linha 2 que deverá ser retirada.

Como você pode perceber, o REM tem suas vantagens (por permitir SAVE) mas também tem seus inconvenientes ('76' e '7E'). Apresentaremos agora mais um lugar onde podemos colocar os programas em linguagem de máquina, perfazendo um total de três possibilidades (se incluirmos o REM e RAMTOP); a escolha de onde colocar o programa dependerá de cada caso e de sua preferência.

No próximo capítulo veremos como transferir um programa que está depois de RAMTOP para dentro de uma instrução REM.

A Memória das Variáveis (após VARS)

Para colocar um programa em linguagem de máquina na região ocupada pelas variáveis, basta reservar uma matriz STRING que tenha o tamanho do programa:

DIM M$ (tamanho)

Para garantir que ela fique logo após o endereço indicado por VARS, faça um CLEAR e coloque o DIM como sendo a primeira instrução no programa que "mexe" com variáveis. Se você quiser dimensioná-la diretamente sem estar no programa, lembre-se também de fazer antes um CLEAR. O endereço inicial será então dado por VARS + 6, ou seja, PEEK 16400 + 256*PEEK 16401 + 6, pois antes das variáveis em si é guardado o nome da matriz, o STRING, o número

total de elementos mais um para o número de dimensões e o número de dimensões (ver o manual do TK).

Note, entretanto, que, se você não tiver expansão, o valor de VARS varia continuamente durante a execução do programa em BASIC e você deve calcular sempre o endereço inicial do seu programa em linguagem de máquina, o que pode ser meio tedioso.

Como sempre, há um cuidado a ser tomado: ao fazer RUN ou CLEAR o seu programa será "apagado", pois está na região das variáveis; portanto, lembre-se sempre de usar GOTO 1 em vez de RUN.

HEXAMEM II

Apresentaremos agora uma segunda versão do HEXAMEM, chamada HEXAMEM II, que vai permitir a entrada de dados não só na forma hexadecimal mas também com números decimais ou caracteres.

```
  1 REM FLAVIO ROSSINI
  5 REM  HEXAMEM II
 10 SLOW
 15 PRINT "MEMORIA INICIAL ?"
 20 INPUT I$
 25 IF I$(1)<>"." THEN GOTO 35
 27 LET K=VAL I$(2 TO )
 30 GOTO 50
 35 LET X$=I$
 40 GOSUB 1500
 45 LET K=T
 50 PRINT AT 0,16;"=.";K
 55 LET INI=K
 60 LET A$=""
 70 IF A$="" THEN INPUT A$
 72 IF A$="XS" THEN GOTO 255
 75 IF A$="XF" THEN GOTO 250
 80 IF A$(1)="." THEN GOTO 140
 85 IF A$(1)=";" THEN GOTO 175
 90 IF A$="P" THEN STOP
 92 IF A$="XX" THEN GOTO 265
 95 LET AUX=16*CODE A$+CODE A$(
2 -476
 100 POKE K,AUX
 105 LET T=K
 110 GOSUB 1000
 115 SCROLL
 120 PRINT "M.";K;TAB 9;X$;TAB 1
5;A$( TO 2);TAB 18;".";AUX;TAB 2
3;";";CHR$ AUX
 125 LET K=K+1
 130 LET A$=A$(3 TO )
```

```
135 GOTO 70
140 IF A$(2)="D" THEN LET T=VAL
A$(3 TO )
142 IF A$(2)<>"D" THEN LET T=VA
L A$(2 TO )
145 GOSUB 1000
150 IF T>255 THEN LET A$( TO 2)
=X$(3 TO 4)
155 IF T>255 THEN LET A$(3 TO 4
)=X$( TO 2)
160 IF T>255 THEN LET A$=A$( TO
4)
166 IF T<=255 AND A$(2)<>"D" TH
EN GOTO 171
167 IF T<=255 AND A$(2)="D" AND
X$="" THEN LET X$="0"
168 IF T<=255 AND A$(2)="D" AND
LEN X$=1 THEN LET X$="0"+X$
169 IF T<=255 AND A$(2)="D" THE
N LET A$=X$+"00"
170 GOTO 95
171 IF X$="" THEN LET X$="0"
172 IF LEN X$>1 THEN LET A$=X$
173 IF LEN X$=1 THEN LET A$="0"
174 GOTO 170
175 LET A$=A$(2 TO )
180 IF A$="" THEN GOTO 70
185 LET T=K
190 GOSUB 1000
195 LET Y$=X$
200 LET T=CODE A$
205 LET W$=A$(1)
210 IF T=221 THEN LET W$=" N.L.
"
215 IF T=221 THEN LET T=118
220 POKE K,T
225 GOSUB 1000
226 IF X$="" THEN LET X$="00"
227 IF LEN X$=1 THEN LET X$="0"
+X$
230 SCROLL
235 PRINT "M.";K;TAB 9;Y$;TAB 1
5;X$;TAB 18;".";T;TAB 23;";";W$
240 LET K=K+1
245 GOTO 175
250 FAST
255 CLS
257 PRINT USR (INI)
260 STOP
265 CLS
270 PRINT "INICIO DA SUBROTINA
?"
```



```
275 GOTO 20
1000 FAST
1005 LET X$=""
1010 LET J=0
1015 LET F=0
1020 LET P=T
1025 FOR I=1 TO 4
1030 LET R=INT (P/16**(4-I))
1035 IF R=0 AND F=0 THEN GOTO 10
60
1040 LET J=J+1
1045 LET F=1
1050 LET X$=X$+CHR$ (R+28)
1055 LET P=P-R*16**(4-I)
1060 NEXT I
1065 SLOW
1070 RETURN
1500 FAST
1505 LET L=LEN X$
1510 LET T=0
1515 LET E=L-1
1520 FOR I=1 TO L
1525 LET T=T+(CODE X$(I)-28)*16*
*E
1530 LET E=E-1
1535 NEXT I
1540 SLOW
1545 RETURN
```

Prog. 8.8 – HEXAMEM II.

Observação: Em alguns casos será conveniente substituir a linha 257 por:

257 RAND USR INI

Em exemplos futuros, você "sentirá" a necessidade desta alteração!

Esse programa usa a seguinte convenção para distinguir números decimais de hexadecimais e de caracteres:

a) números hexadecimais: são introduzidos *normalmente;*
b) números decimais: *devem* começar com um ponto (.);
c) caracteres: *devem* ser precedidos por um ponto e vírgula (;).

Desse modo, ao iniciar o programa, ele pergunta "memória inicial = ?" e, supondo que ela seja 30000, você deverá responder:

.30000 (decimal)

ou

7530 (hexadecimal)

Agora o HEXAMEM II está aguardando dados para colocar na memória. Suponha que tivéssemos o seguinte programa onde usamos para identificar os caracteres um *traço*, sublinhando-os:

```
        LD   HL,(D-FILE) ; carrega HL com o endereço do início da
                         ;   tela que é um NEW LINE
        INC  HL          ; incrementa HL para indicar a primeira
                         ;   posição real da tela
        LD   B,22        ; B = 22 para contador de linhas
        LD   C,31        ; C = contador de colunas para *
LOOPL   LD   A,*         ; coloca em A o código de *
        LD   D,C         ; coloca C em D para preservar o valor de C
                         ;   durante o loop
        LD   E,32        ; E = contador de colunas para [*]
LOOP    LD   (HL),A      ; coloca * na tela D vezes
        INC  HL
        DEC  E
        DEC  D
        JP NZ, LOOP
        LD   A, [*]      ; coloca em A o código de [*]
LOOPC   LD   (HL),A      ; coloca [*] na tela (E-D) vezes
        INC  HL
        DEC  E
        JP NZ, LOOPC
        INC  HL
        DEC  C
        DJNZ  LOOPL      ; decrementa B e volta para imprimir a pró-
                         ;   xima linha
        RET
```

Prog. 8.9 – Programa para encher a tela com asteriscos normais e em vídeo reverso.

Para colocá-lo no computador, basta fazer:

```
"2A.16396"   (N.L.)   (note que para números decimais o HEXAMEM II
                       faz automaticamente a inversão)
"23"         (N.L.)
"06.22"      (N.L.)
"0E.31       (N.L.)
"3E; *"      (N.L.)   (o (;) indica caractere)
"51"         (N.L.)
"1E.32"      (N.L.)
"77"         (N.L.)
"23"         (N.L.)   Observação: LOOPL = 30008
"1D"         (N.L.)               LOOP  = 30013
"15"         (N.L.)               LOOPC = 30022
"C2.30013"   (N.L.)
"3E; [*] "   (N.L.)   [*] = asterisco em vídeo-reverso
"77"         (N.L.)
"23"         (N.L.)
"1D"         (N.L.)
"C2.30022"   (N.L.)
"23"         (N.L.)
"0D"         (N.L.)
"10.232"     (N.L.)   Observação: –24 = 'E8' = +232
"C9"         (N.L.)
```

Feito isso, se você teclar *P* o programa pára. Se você desejar executar o programa basta, como de costume, introduzir XS (SLOW) ou XF (FAST).

Execute então o programa e observe o efeito e a velocidade (mesmo em SLOW...). Você saberia explicar como esse programa funciona?

Note que o HEXAMEM II faz automaticamente a inversão para números decimais maiores que 255 e menores que 65536 (basta observar a memória na tela para instruções do tipo LD HL, 16396 = 2A.16396). Tome cuidado, entretanto, pois essa inversão *não* é feita para os números hexadecimais! Além disso, se for carregar pares de registros ou executar algum JUMP absoluto ou CALL com números menores que 256, você *deve* informar ao HEXAMEM II que o número tem 2 bytes: basta acrescentar um *D* na frente do número. Por exemplo:

LD BC,30 ⇨ 01.D30 e *não* 01.30
CALL 200 ⇨ CD.D200 e *não* CD.200

Quanto aos caracteres, note que o caractere <> (tecla T) será sempre substituído pelo código 118 (NEW LINE) e um N.L. aparecerá no canto direito da tela. No próximo capítulo falaremos com detalhes sobre a utilidade desse NEW LINE.

Nos programas, usaremos um traço para sublinhar os caracteres; assim:

LD A,B̲ (3E;B)

significa: coloque em A o código do caractere B e *não* copie o registro B em A (isto seria simplesmente LD A,B).

Note que o HEXAMEM II fornece como saída na tela o endereço e o conteúdo da memória em decimal (precedidos por um ponto), o endereço e o conteúdo da memória em hexadecimal e, finalmente, o caractere que corresponde ao código do que está na memória precedido por um ponto e vírgula (tudo equivalente à convenção adotada).

Além disso, ele permite que a memória "de partida" da sub-rotina seja modificada: para isto basta fazer XX em vez de XF, XS ou P e colocar o endereço "de partida" da sub-rotina. Feito isto, pode-se então mandar executar o programa (XS ou XF). O porquê deste procedimento será detalhado no próximo capítulo.

Resumo

Neste capítulo estudamos a organização da memória ROM e RAM para o TK com 16 K de expansão, destacando algumas sub-rotinas que executam os comandos em BASIC que estão na memória ROM e as várias regiões da memória RAM, a saber: as variáveis do programa na ROM, a memória do programa em BASIC, a tela de TV, as variáveis do programa em BASIC, espaço para a linha sendo digitada e os vários STACKs. Essas regiões têm suas fronteiras indicadas por variáveis que estão entre as memórias 16384 e 16508, que seriam: D-FILE, VARS, E-LINE, STKBOT, STKEND, ERR-SP e RAMTOP.

Aprendemos a maneira como são armazenados na memória de programa as linhas e as variáveis numéricas (para as matrizes, loops de FOR/NEXT etc. basta olhar o manual do TK), e um método para economizar memória utilizando STRINGs ao invés de números.

A seguir, vimos como colocar os programas em linguagem de máquina de maneira a poderem ser colocados na fita usando SAVE (apesar de serem agora "apagáveis" pelo NEW), ou seja, em outro lugar que *não seja após* RAMTOP. Assim, temos agora três opções de lugares para colocarmos os programas:

a) num REM: desde que tomemos os devidos cuidados com os números '76' e '7E';
b) no fim da memória RAM, ou seja, após RAMTOP, sendo que o programa fica imune ao NEW mas também ao SAVE;
c) na memória das variáveis do programa BASIC, desde que não usemos RUN ou CLEAR e tomemos cuidado se não tivermos memória de 16 K.

Finalmente, vimos a segunda versão de HEXAMEM, o HEXAMEM II, que aceita dados decimais, hexadecimais ou caracteres.

Exercícios

1. Para recordar as instruções, faça um programa que, por subtrações sucessivas, calcule o quociente e o resto da divisão de um número que está no par DE por um número que está no registro L, deixando a resposta no par BC e o resto no registro H.
2. Faça um programa em BASIC que seja capaz de copiar o conteúdo da tela em uma variável STRING. Assim, para obter o conteúdo da tela copiado, bastará fazer um PRINT "variável STRING que contém a tela".
   Sugestão: Você sabe o endereço da tela (D-FILE) e o das variáveis (VARS). Faça um CLEAR para garantir que sua variável fique no começo da região indicada por VARS e transfira a tela para a primeira variável que estiver nessa região. Lembre-se que a variável deverá ser STRING...
3. Explique o que ocorre no programa 8.9 se invertermos a 10ª e 11ª instruções, ou seja, colocarmos antes DEC D e depois DEC E.
4. Um bom "truque" para colocar nomes em seus programas de maneira que *ninguém* possa retirá-los é colocar uma linha Ø com o seu nome. Para isto, basta fazer, por exemplo:

   1 REM FLAVIO ROSSINI

   NEW LINE, e, a seguir:

   ```
   POKE 1651Ø,Ø
   NEW LINE
   NEW LINE
   ```

   Tente agora apagar ou alterar a linha Ø que apareceu, usando os métodos convencionais. Pergunta-se:
   a) Por que a linha Ø aparece no lugar da linha 1?
   b) Como retirá-la?
   c) Como criar mais linhas Ø?
   d) Poderíamos criar duas ou mais linhas número 1 (ou qualquer outro número)? Como? Neste caso, o que ocorre se tivermos três linhas número 1 e fizermos GOTO 1 ou LIST 1?

Observação: Como você deve ter notado (programa 8.3), o TK *não* utiliza a convenção normal para armazenar o número das linhas em BASIC, ou seja, ele armazena antes o byte menos significativo! Esta é a única exceção.

"Brincando" com a Tela de TV

REVISÃO DE D-FILE: O PROGRAMA LABIRINTO

Vamos apresentar agora um programa (em BASIC) bastante interessante, feito por Ricardo de Fayetti Siqueira, que utiliza amplamente as instruções PEEK e POKE para ilustrar o funcionamento da tela.

Como você deve lembrar, a região de memória correspondente à tela está logo após a região onde se armazena(m) o(s) programa(s) em BASIC e a variável que mostra seu começo (D-FILE) está guardada nos endereços 16396 e 16397; assim, uma instrução POKE feita a partir de (D-FILE + 1) até (D-FILE + 792) colocará um caractere diretamente na tela.

A memória da tela ocupa 793 bytes, correspondentes a 24 linhas de 33 caracteres (o 33º é um NEW LINE) mais o primeiro caractere "da tela" que é um NEW LINE (24 x 33 + 1 = 793).

O programa desenha várias linhas horizontais com espaços em branco; você pode controlar, usando as teclas 5, 6, 7 e 8 (ou o joystick), o caractere "I" que aparecer no canto superior esquerdo e seu objetivo é chegar no caractere "▒", cuja posição é gerada aleatoriamente. Você pode passar pelos espaços em branco mas *não* pode passar através da linha preta.

```
   1 REM ** LABIRINTO **
   5 RAND
  10 FAST
  15 CLS
  20 POKE 16418,0
  25 PRINT "████████████████████████████████"
  30 PRINT AT 23,0;"████████████████████████████████"
  35 FOR L=2 TO 22 STEP 2
  40 PRINT AT L,0;"████████████████████████████████"
  45 LET K=INT (31*RND)
  50 PRINT AT L;K;" "
  55 NEXT L
  65 SLOW
  70 LET R=34+PEEK 16396+256*PEEK 16397
  75 POKE R,46
  80 LET H1=INT (RND*756+35)
  85 LET H=H1+R
  90 IF PEEK H=128 OR PEEK H=118 THEN GOTO 80
  95 POKE H,8
 100 LET W=R
 105 IF INKEY$="5" THEN GOTO 200
 110 IF INKEY$="8" THEN GOTO 250
 115 IF INKEY$="6" THEN GOTO 300
 120 IF INKEY$="7" THEN GOTO 350
 125 GOTO 105
 200 POKE W,0
 205 LET W=W-1
 210 IF W=H THEN GOTO 1500
 215 IF PEEK W<>0 THEN LET W=W+1
 220 POKE W,46
 225 GOTO 105
 250 POKE W,0
 255 LET W=W+1
 260 IF W=H THEN GOTO 1500
 265 IF PEEK W<>0 THEN LET W=W-1
 270 POKE W,46
 275 GOTO 105
 300 POKE W,0
 305 LET W=W+33
 310 IF W=H THEN GOTO 1500
 315 IF PEEK W<>0 THEN LET W=W-33
 320 POKE W,46
 325 GOTO 105
 350 POKE W,0
 355 LET W=W-33
 360 IF W=H THEN GOTO 1500
 365 IF PEEK W<>0 THEN LET W=W+33
 370 PRINT W,46
 375 GOTO 105
1500 PRINT AT 10,3;"PRESSIONE ALGUMA TECLA ..."
1505 POKE W,45
1510 POKE W,174
1515 IF INKEY$="" THEN GOTO 1505
1520 PAUSE 100
1525 GOTO 10
```

Prog. 9.1 – Programa LABIRINTO.

Experimente então "brincar" um pouco com o programa! Note que ele usa as linhas 22 e 23... Para isso, basta "enganar" o computador fazendo um POKE 16418,Ø. Em 16418 está uma variável do programa interpretador, chamada DF-SZ, que indica quantas linhas são reservadas para a edição (normalmente essa variável contém o número 2). Entretanto, esse "truque" só funciona para efeitos de tela. Se você tentar fazer um INPUT após este POKE, "coisas estranhas" poderão ocorrer. Imagine como seria lento e complicado esse mesmo programa escrito *totalmente* sem usar PEEK e POKE...

Perceba que a memória é uma pilha de registros e, portanto, devemos pensar de modo linear para escrever o programa em linguagem de máquina (em BASIC, pensamos na tela como se fosse um plano, definindo linhas e colunas, e esse conceito deve ser, em parte, abandonado).

Na memória, a tela divide-se em 24 segmentos que são colocados na TV um embaixo do outro para formar o plano visto por nós.

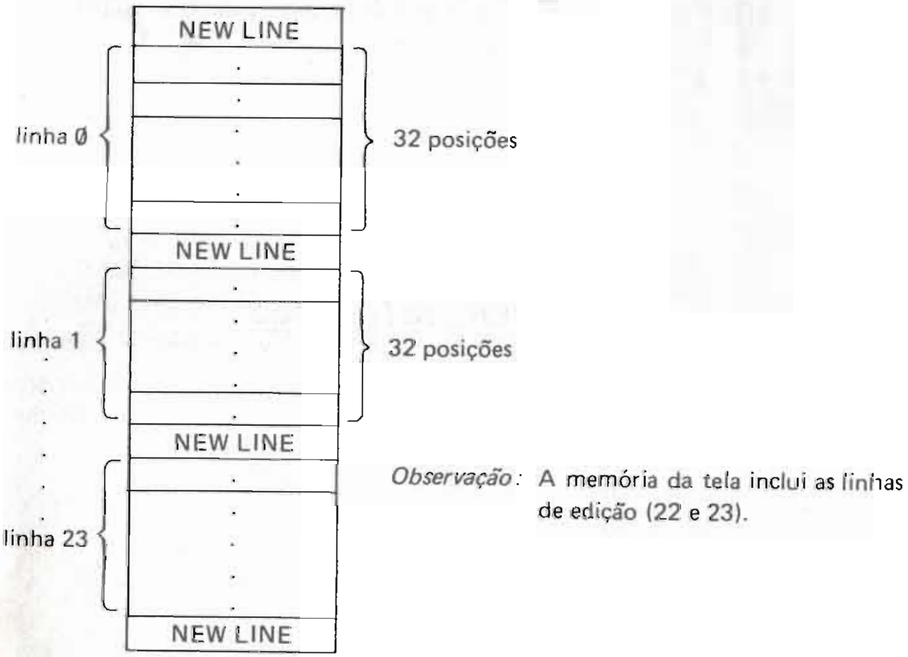


Fig. 9.1 – A tela na memória.

Note que, se quisermos ir uma posição para a *direita*, temos de *incrementar* de uma unidade o endereço da memória onde está o "I", mandar imprimir (usando POKE...) um número que corresponda ao caractere desejado (no caso "I"), e, obviamente, colocar na posição anterior o caractere "branco" (código 'ØØ'). Do mesmo modo, se quisermos ir para a *esquerda* deveremos *decrementar* o endereço da memória. Para ir para *cima* devemos *diminuir de 33 unidades* o endereço e para *descer* devemos *aumentar de 33 unidades* o endereço. (Você poderia explicar por quê?)

Para impedir a passagem pela linha preta, basta verificar se na nova posição de "I" existe o caractere " █ " (código 128) e, em caso afirmativo, inibir o movimento. Também *não* podemos invadir os NEW LINEs (código 118) pois, caso contrário, o computador se perde por não saber "construir" a tela de TV sem eles.

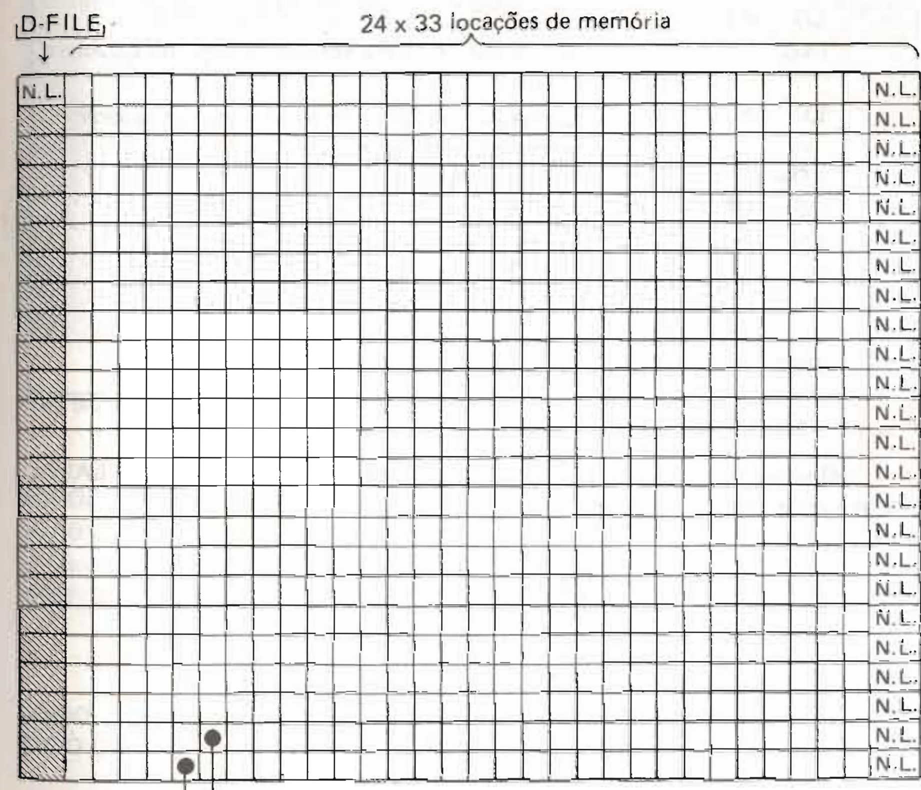


Fig. 9.2 – Tela de TV.

Estude com detalhes o programa LABIRINTO e tente se localizar na tela utilizando a figura 9.2 para ajudá-lo. Note que se não tivéssemos a primeira e última linhas você poderia invadir o programa ou as variáveis... O que aconteceria se colocássemos um número maior que 2 na memória 16418?

O SCROLL HORIZONTAL

Você está lembrado do capítulo 4 e do ANTISCROLL? Experimente agora o seguinte programa:

```
     LD   A,22        ; usa A como contador para as 22 linhas
     LD   HL,(16396)  ; carrega HL com D-FILE
LOOP INC  HL          ; incrementa HL para apontar o primeiro caractere
                        da tela
     LD   D,H         ; transfere HL para DE
     LD   E,L
     INC  HL          ; incrementa HL para apontar o segundo caractere
                        da tela
     LD   BC,31       ; carrega BC com 31 (número de caracteres a serem
                        deslocados para esquerda)
     LDIR             ; desloca os caracteres
     DEC  HL
     LD   (HL),Ø      ; coloca "branco" na última coluna
     INC  HL
     DEC  A           ; decrementa contador
     JP   NZ, LOOP    ; se não for zero repete para a próxima linha
     RET
```

Prog. 9.2 – SCROLL horizontal.

Coloque-o a partir da memória 3ØØØØ usando HEXAMEM II; o LABEL LOOP corresponde ao endereço 3ØØØ5. Assim, basta fazer:

```
"3E.22"       (N.L.)
"2A.16396"    (N.L.)
"23"          (N.L.)
"54"          (N.L.)
"5D"          (N.L.)
"23"          (N.L.)
"Ø1.D31"      (N.L.)
"EDBØ"        (N.L.)
"2B"          (N.L.)
"36; ƀ"       (ƀ = espaço em branco)
"23"          (N.L.)
"3D"          (N.L.)
"C2.3ØØØ5"    (N.L.)
"C9"          (N.L.)
"P"           (N.L.)
```


Execute então o seguinte programa em BASIC que demonstra o efeito de SCROLL *para a esquerda*. O início do programa, apesar de ser em FAST, é um pouco demorado pois ele deve gerar 7Ø4 caracteres aleatórios...

```
2005 REM ** SCROLL LATERAL **
2010 FAST
2015 RAND
2020 LET A$=""
2025 FOR I=1 TO 704
2030 LET X=INT (2*RND)
2035 LET A$=A$+CHR$ (INT (64*RND
) +X*128)
2040 NEXT I
2045 SLOW
2050 PRINT A$
2055 FOR I=1 TO 22
2060 RAND USR 30000
2065 NEXT I
2070 CLS
2075 GOTO 2050
```

Prog. 9.3 – Utilização do SCROLL horizontal.

Que tal o efeito? Antes de prosseguir, tente entender detalhadamente o SCROLL horizontal (se necessário, estude os programas anteriores do ANTI-SCROLL, capítulos 3 e 5, e SCROLL, capítulo 3). O programa transfere para a esquerda linha por linha da tela utilizando a instrução LDIR. Você consegue perceber isto "visualmente"? Por que é necessária a instrução CLS na linha 2Ø7Ø?

COMO TRANSFERIR PROGRAMAS EM LINGUAGEM DE MÁQUINA DO TOPO DA RAM PARA DENTRO DE UM REM

Caso você deseje gravar numa fita esse programa de SCROLL horizontal, basta transferi-lo para dentro de uma linha de REM. Você pode, inclusive, retirar o HEXAMEM II do computador usando NEW (desde que você tenha se lembrado de reservar memória modificando RAMTOP). Assim, digite NEW e reserve espaço num REM para que caiba o programa de SCROLL lateral:

```
1 REM Ø123456789Ø123456789Ø1234
```

A seguir, acrescente o seguinte programa:

```
   2 REM "TRANSFMEM"
   5 PRINT "MEMORIA FONTE ?"
  10 INPUT F
  15 PRINT
  20 PRINT "MEMORIA DESTINO ?"
  25 INPUT D
  30 PRINT
  35 PRINT "TAMANHO DO PROGRAMA
EM BYTES ?"
  40 INPUT T
  45 FAST
  50 FOR I=1 TO T
  55 POKE D,PEEK F
  60 LET D=D+1
  70 LET F=F+1
  75 NEXT I
  80 PRINT
  85 PRINT "TRANSFERENCIA COMPLE
TA"
```

Prog. 9.4 – TRANSFMEM.

Execute o programa e coloque então memória fonte = 30000, memória destino = 16514 e tamanho do programa em bytes = 23. Agora você pode retirar o programa TRANSFMEM linha por linha (não use NEW) e, a seguir, salvar o REM que contém o programa SCROLL lateral:

SAVE "HORSCR" (N.L.)
(poderia ser qualquer coisa)

Isso logicamente é válido para qualquer programa que estiver após RAMTOP.

Observação: Para executar o programa agora é necessário fazer RAND USR 16514 e *não* 30000.

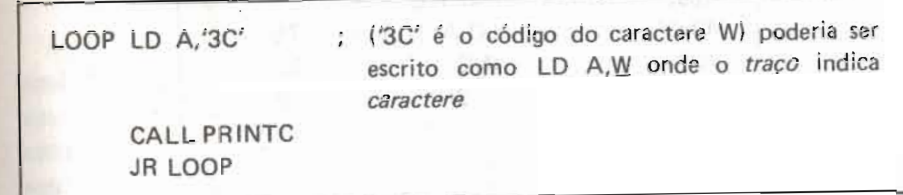
AS SUB-ROTINAS DA ROM E A TELA DE TV – AS PSEUDO-INSTRUÇÕES

Como vimos no capítulo anterior, já existe na ROM uma sub-rotina que imprime um caractere na tela, cujo endereço é '0808'. Essa sub-rotina coloca na

primeira posição livre que houver na tela o caractere cujo código está no acumulador. Chamaremos essa sub-rotina de PRINTC (PRINT Character) e faremos um pequeno programa para chamá-la. Usando HEXAMEM II, coloque-a a partir da memória 16514. Para isso, reserve espaço no início da memória de programa com uma instrução REM:

1 REM 0123456789

Eis o programa:



```
LOOP LD A,'3C'     ; ('3C' é o código do caractere W) poderia ser
                     escrito como LD A,W onde o traço indica
                     caractere
     CALL PRINTC
     JR LOOP
```

Prog. 9.5 – Programa para colocar o caractere W na tela.

que no programa HEXAMEM II ficaria:

"3E;W" (N.L.)
"CD0808" (N.L.)
"18F9" (N.L.)

Execute-o em FAST (XF) e, a seguir, coloque o computador em SLOW e repita o programa fazendo RAND USR 16514. Não se preocupe com a falta da instrução RET: *nesse caso*, o programa volta automaticamente ao BASIC ao terminar a tela. Repare na velocidade nos *dois* casos. A título de comparação, execute o seguinte programa BASIC em SLOW e FAST:

```
2000 PRINT "W";
2005 RUN 2000
```

Prog. 9.6 – Programa 9.5 em BASIC.

Como você faria, no programa 9.5, para trocar o caractere usando POKE?

Vamos agora implementar a sub-rotina para que ela imprima qualquer *mensagem* e não apenas um caractere. Para tanto, necessitamos de um código para indicar ao computador "fim de mensagem". Utilizaremos o número '43', já que ele não é código de nenhum caractere (ver apêndice 2).

Vamos também adicionar ao nosso vocabulário de mnemônicos mais duas pseudo-instruções:

DC = defina constante
DM = defina mensagem

Elas são chamadas de pseudo-instruções por não terem correspondente em código hexadecimal, ou seja, servem apenas para tornar mais clara a escrita de um programa. Observe então o seguinte exemplo:

início de um REM na linha 1, ou seja, 16514

```
'4Ø82'  PRINTM   POP HL          'E1'
'4Ø83'           LD A,(HL)       '7E'
'4Ø84'           INC HL          '23'
'4Ø85'           PUSH HL         'E5'
'4Ø86'           CP '43'         'FE43'
'4Ø88'           RET Z           'C8'
'4Ø89'           CALL PRINTC     'CDØ8Ø8'
'4Ø8C'           JR PRINTM       '18F4'
```

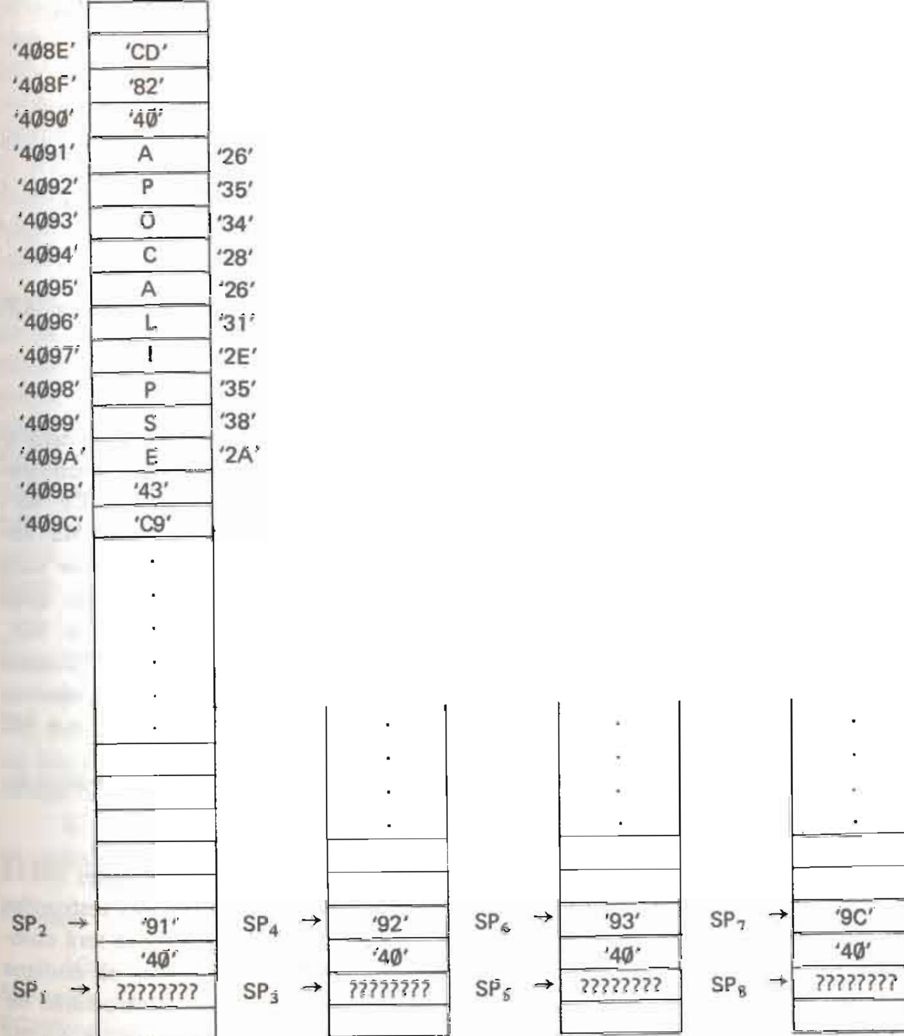
Prog. 9.7 – PRINTM (PRINT Message).

e suponha que, a seguir, façamos:

```
'4Ø8E'  CALL  PRINTM       'CD824Ø'
'4Ø91'  DM    APOCALIPSE   ;APOCALIPSE
'4Ø9B'  DC    '43'         '43'
'4Ø9C'  RET                'C9'
```

Primeiramente repare que as pseudo-instruções DM e DC não têm correspondente em código hexadecimal; na memória é colocado apenas o código dos caracteres da mensagem ou da constante. Nosso programa terá como endereço inicial a memória '4Ø8E' (.16526). Você se lembra como funciona a instrução CALL? Ela faz um PUSH (ou seja, coloca no STACK) com o *endereço de retorno* da sub-rotina e, a seguir, carrega o PC com o *endereço inicial* da sub-rotina que está sendo chamada. Logo, a letra A da mensagem APOCALIPSE, que fica após o CALL PRINTM, está no endereço indicado pelo topo do STACK, após ser execu-

corresponde ao endereço de retorno de PRINTM. Assim, após CALL PRINTM, PC passa a indicar '4Ø82' e, ao fazer POP HL, no par HL teremos o endereço da letra A e basta então fazer LD A, (HL) para colocar o código da letra A no acumulador, a qual será colocada na tela por PRINTC (pois seu código é diferente de '43').



PC = '4Ø8E' SP_1
PC = '4Ø82' ← CALL PRINTM/SP_2
PC = '4Ø83' ← POP HL/SP_3 (H = '4Ø' ; L = '91')
PC = '4Ø85' ← INC HL (H = '4Ø' ; L = '92')
PC = '4Ø86' ← PUSH HL/SP_4

.
.

PC = '4Ø83' ← POP HL/SP_5 (H = '4Ø' ; L = '92')

.
.

PC = '4Ø85' ← INC HL (H = '4Ø' ; L = '93')
PC = '4Ø86' ← PUSH HL/SP_6

.
.
(H = '4Ø' ; L = '9B')
PC = '4Ø85' ← INC HL (H = '4Ø' ; L = '9C')
PC = '4Ø86' ← PUSH HL/SP_7
PC = '4Ø88' ← CP '43' (agora H = '4Ø' L = '9C' e A = '43')
PC = '4Ø88' — (flag Z = 1) — RET Z/SP_8

Fig. 9.3 – Explicação da lógica (SP).

Ao incrementar HL e fazer um PUSH HL, o topo do STACK conterá o endereço da letra P (Por quê?). Basta então repetir o processo até encontrar o número '43', que indica fim de mensagem; o que é feito pela instrução CP '43', que testa a igualdade entre o acumulador e o número '43'. Se o número '43' for encontrado, a sub-rotina PRINTM terminará (RET Z), voltando o controle para o endereço indicado pelas memórias do topo do STACK, ou seja, logo após o número '43' (no caso um RET para o BASIC; em outras palavras, se A = '43', teremos PC = '4Ø9C'); caso contrário, basta chamar a sub-rotina PRINTC para imprimir o caractere que está no acumulador. Para maior entendimento, observe a figura 9.3, que indica a seqüência das instruções, o andamento do SP e o PC gerado pelas instruções.

Vamos então colocar esse programa na memória através de uma instrução REM. Faça:

1 REM Ø123456789Ø123456789Ø123456

para reservar os 27 bytes necessários; a seguir, use HEXAMEM II, colocando 16514 ('4Ø82') para endereço inicial; introduza os códigos hexadecimais das instruções e, ao chegar em APOCALIPSE, introduza (;APOCALIPSE) e a palavra será colocada na memória diretamente, não havendo necessidade de colocar os códigos hexadecimais de cada letra. Veja como o HEXAMEM II facilita a introdução de mensagens. Ao terminar de introduzir o programa na memória, *não* digite XS ou

XF, pois o início do programa está, na realidade, na memória 16526. Assim, digite XX e o HEXAMEM II perguntará qual o endereço inicial da sub-rotina. Você responde com .16526 (NEW LINE) e, a seguir, pode digitar XS ou XF à sua escolha... Tente explicar os números que aparecem ao lado de "APOCALIPSE"... Experimente ordenar: RAND USR 16526. (Ver página 139.)

Você deve estar lembrado que o HEXAMEM II possibilita a introdução de NEW LINEs cada vez que pressionamos <> (SHIFT-T) em uma cadeia de caracteres, o que nos possibilita "pular linhas" ou indicar "fim de linha" na tela em um programa em linguagem de máquina, pois o código de NEW LINE (118) é interpretado como fim de linha pelo computador. Assim, vamos fazer uma pequena modificação no programa anterior acrescentando mais algumas mensagens após a pseudo-instrução DM. Devemos, entretanto, aumentar o espaço reservado pelo REM. (*Cuidado! Não* use EDIT, pois o byte '7E' faz parte do programa.) Use então as técnicas mencionadas no capítulo anterior para aumentar o REM juntando uma linha 2 de pelo menos 6Ø bytes. Feito isso, você poderá modificar o fim do programa utilizando o HEXAMEM II para introduzir mensagens a partir da memória 16529 ('4Ø91'), que corresponde ao começo da palavra APOCALIPSE. Faça então:

```
MEM. 16529  DM   ALL<>                    ; ALL <>
MEM. 16533  DM   ƀ YOU<>                  ;   YOU <>
MEM. 16538  DM   ƀƀ NEED<>                .
MEM. 16545  DM   ƀƀƀ IS<>                 .
MEM. 16551  DM   ƀƀƀƀ LOVE<>              .
MEM. 1656Ø  DM   <>                       .
MEM. 16561  DM   LENNON/MCCARTNEY<>       .
MEM. 16578  DM   1968                     ; 1968
MEM. 16582  DC   '43'                     '43'
MEM. 16583  RET                           'C9'
```

Note que basta você dizer ao HEXAMEM II que o endereço inicial é '4Ø91' (16529) (você saberia explicar por quê?) e introduzir a mensagem literalmente, começando com um ponto e vírgula (;) e incluindo os <> (SHIFT-T), que indicam fim de linha. Terminada a mensagem, basta colocar os hexadecimais '43' (fim de mensagem) e 'C9' (RET). A seguir, novamente digite XX e o endereço inicial da sub-rotina (.16526) seguido de XS ou XF (ou RAND USR 16526...).

É conveniente salientar que a sub-rotina PRINTC destrói *todos* os pares de registros; assim, lembre-se de salvá-los, se for necessário.

Resumo

Vimos neste capítulo como utilizar PEEK e POKE para aumentar a *performance* de um programa BASIC que utiliza a tela de TV, auxiliados por um POKE na variável DF-SZ (memória 16418) que aumenta o tamanho da tela.

Feito isto, apresentamos um programa que faz um SCROLL horizontal da tela usando os princípios já estudados para o SCROLL e ANTISCROLL.

Aprendemos a utilizar a sub-rotina da ROM de endereço 'Ø8Ø8' (PRINTC) para colocar mensagens na tela, ajudados pelo caractere de NEW LINE, e fizemos uma comparação de velocidades entre o BASIC e a linguagem de máquina.

Você pode utilizar esse mesmo procedimento para imprimir campos (ou tabuleiros) que podem ser usados em jogos gráficos *sem* utilizar o BASIC.

Exercícios

1. Explique todos os programas em BASIC apresentados neste capítulo.
2. Modifique o programa LABIRINTO para que o alvo mude de posição após certo tempo, se você ainda não o atingiu. No fim do programa, ao atingir o alvo, o computador deve imprimir o tempo que você demorou para atingir o alvo com os respectivos comentários sobre sua *performance*.
3. Modifique o programa do SCROLL horizontal para que ele faça um SCROLL para a direita.
4. Unindo os conceitos de SCROLL vertical e SCROLL horizontal, faça um programa que realize um SCROLL em diagonal.
5. Utilize as sub-rotinas PRINTM e PRINTC para encher a tela com caracteres a sua escolha e use o par DE como contador para indicar que a tela foi preenchida (DE = 7Ø4). A seguir, dê uma pause e simule um CLS com sub-rotinas em *linguagem de máquina*. Observações quanto ao problema:
   a) Ao usar pares de registro como contadores,existe um pequeno problema: a instrução DEC pares de registros *não* afeta a flag Z (apêndice 2); portanto, você terá de usar duas instruções CP para saber se o par DE atingiu o valor Ø (caso você use DEC e valor inicial 7Ø4) ou o valor 7Ø4 (caso você use INC e valor inicial Ø).
   b) Uma pausa pode ser obtida com um loop que não faz nada.
   c) O CLS pode ser simulado enchendo a tela de brancos.
6. Como você faria um programa como o TRANSFMEM (programa 9.4) em linguagem de máquina?
7. Baseado no programa 4.8 e utilizando os conceitos vistos neste capítulo de SCROLL para a esquerda mais os exercícios 3 e 4, faça um programa capaz de "rolar" a tela em quatro direções:
   - vertical ↕
   - horizontal ↔
   - diagonal principal ↘
   - diagonal secundária ↗

Entendendo o Teclado

SUB-ROTINAS DA ROM PARA DECODIFICAÇÃO DO TECLADO – A FUNÇÃO INKEY$ EM LINGUAGEM DE MÁQUINA

Para poder utilizar o teclado TK em nossos programas, devemos primeiramente entender duas sub-rotinas existentes na ROM que são capazes de detectar se alguma tecla foi pressionada e, em caso afirmativo, qual delas, para a seguir associar à mesma o caractere correspondente. A primeira dessas sub-rotinas está no endereço 'Ø2BB' e serve para detectar qual tecla foi pressionada; chamaremos essa sub-rotina de KDEC (Key DEteCtion).

O teclado é dividido internamente no computador em setores verticais e horizontais, numerados de Ø a 7, conforme a figura 1Ø.1.

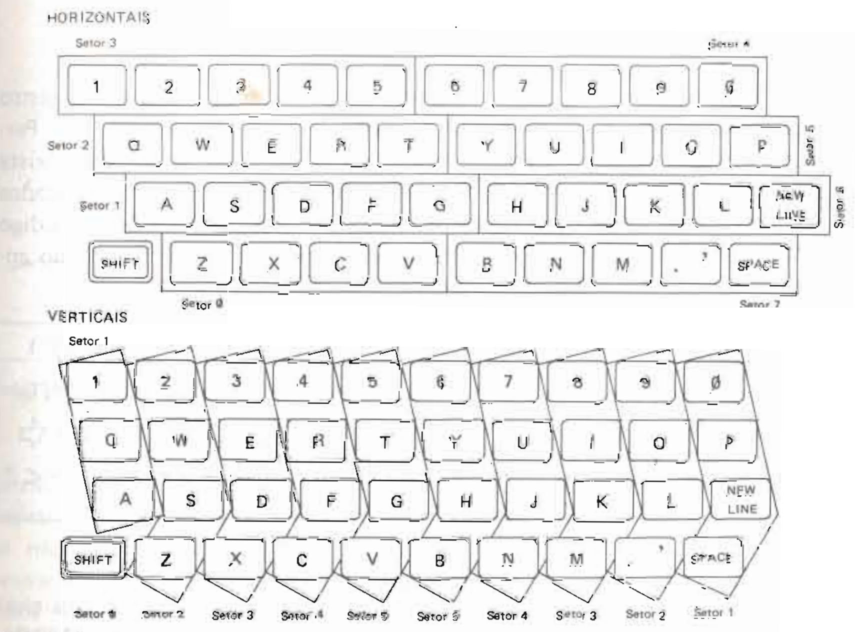


Fig. 1Ø.1 – Divisão do teclado em setores.

Como você pode notar, a tecla SHIFT *não* está incluída nos setores horizontais mas tem um setor vertical *exclusivo*. A sub-rotina KDEC coloca no par HL um número que corresponde à tecla pressionada, da seguinte maneira: o registro L corresponde aos setores horizontais e o H aos verticais (como sabemos, cada registro tem 8 bits que podem ser numerados de Ø a 7). Desse modo, se nenhuma tecla for pressionada, ambos os registros retornarão com o valor 'FF' ("11111111"); caso contrário, o bit correspondente ao setor (de Ø a 7) será colocado em zero. Por exemplo, se você apertar a letra K (horizontal = 6, vertical = 3), o par HL conterá o valor "1111Ø111 1Ø111111". Resumindo:

Setor Vertical	Registro H	Hexa	Setor Horizontal	Registro L	Hexa
nenhuma tecla	"11111111"	'FF'	nenhuma tecla	"11111111"	'FF'
setor Ø	"1111111Ø"	'FE'	setor Ø	"1111111Ø"	'FE'
setor 1	"111111Ø1"	'FD'	setor 1	"111111Ø1"	'FD'
setor 2	"11111Ø11"	'FB'	setor 2	"11111Ø11"	'FB'
setor 3	"1111Ø111"	'F7'	setor 3	"1111Ø111"	'F7'
setor 4	"111Ø1111"	'EF'	setor 4	"111Ø1111"	'EF'
setor 5	"11Ø11111"	'DF'	setor 5	"11Ø11111"	'DF'
			setor 6	"1Ø111111"	'BF'
			setor 7	"Ø1111111"	'7F'

Tab. 1Ø.1 – Conteúdo dos registros H e L ao pressionar alguma tecla.

Se você pressionar SHIFT juntamente com alguma tecla, o bit Ø do registro H será zero. Assim, SHIFT J produzirá "111Ø111Ø1Ø111111" no par HL. Percebe-se facilmente que cada tecla produzirá um valor diferente no par HL. Existe então uma outra sub-rotina na ROM (chamada ACHR) que, de posse desses dados colocados no *par BC*, coloca, *em HL*, o endereço da memória onde está o código do caractere desejado. *Esses códigos estão armazenados na ROM* a partir do endereço 'ØØ7E' na seguinte ordem (setores em ordem crescente):

Z X C V A S D F G Q W E R T 1 2 3 4 5 Ø 9 8 7 6 P O I
U Y NEW LINE L K J H SPACE . M N B : ; ? / STOP LPRINT
SLOW FAST LLIST " " OR STEP < = <> EDIT AND THEN TO ⇦
RUBOUT GRAPHICS ⇨ ⇧ ⇩ ") ($ >= FUNCTION = + – **£ , >< *

Tab. 1Ø.2 – Ordem dos códigos armazenados na ROM.

Assim, depois de colocar em BC os valores obtidos por KDEC, basta chamar a sub-rotina ACHR (acha caractere), que está a partir da memória 'Ø7BD'.

Essa sub-rotina coloca no par HL um endereço entre 'ØØ7E' (letra Z) e 'ØØCB' (*) correspondente ao caractere da tecla que foi pressionada. Note, entretanto, que se nenhuma tecla está sendo pressionada você deve detectar isso antes de chamar a sub-rotina ACHR, para não obter caracteres "estranhos" correspondentes a valores prévios de HL. Lembre-se que não existe um comando que corresponderia ao CLEAR em BASIC; assim, os valores iniciais dos registros, ao chamar uma sub-rotina em linguagem de máquina no TK, são desconhecidos, exceção feita ao par BC e ao PC. Desse modo, se você chamar ACHR e não tiver pressionado nenhuma tecla, ela certamente retornará um valor no par HL, mas que corresponde a algum caractere aleatório. Outra coisa a ser comentada é que a sub-rotina KDEC destrói o valor de todos os registros. Assim sendo, se porventura você for utilizá-la em separado, lembre-se de fazer PUSH AF, PUSH BC e PUSH DE antes de chamá-la e POP DE, POP BC e POP AF depois. Observe agora o seguinte exemplo:

```
TECTV CALL  KDEC     'CDBBØ2'  ; detecta a tecla
      LD    B,H      '44'      ; transfere para BC preparando
                                 para ACHR
      LD    C,L      '4D'
      LD    A,C      '79'      ; transfere L para A
      CP    'FF'     'FEFF'    ; testa se há tecla pressionada
      LD    A,Ø      '3EØØ'
      JR Z, TECTV    '28.244'  ; se não houver tecla pressionada,
                                 volta para o início
      CALL  ACHR     'CDBDØ7'  ; se houver, procura o código
      LD    A,(HL)   '7E'      ; coloca código em A
      CALL  PRINTC   'CDØ8Ø8'  ; imprime caractere
      CP    Ø        'FEØØ'    ; se for espaço em branco (código
                                 Ø), volta ao BASIC
      RET Z          'C8'
      JR    TECTV    '18.232'  ; caso contrário, volta e espera
                                 outra tecla
```

Prog. 1Ø.1 – TECTV = TECla/TV.

Note que, após chamar KDEC, se não houver nenhuma tecla pressionada (*incluindo* SHIFT), teremos o valor 'FF' no registro L; portanto, passando esse registro para A e comparando-o com 'FF', será afetada a flag Z que é testada: se não houver nenhuma tecla pressionada, volta para o início; caso contrário, prossegue e acha o caractere. A instrução LD A,Ø foi colocada para fins didáticos (ela não altera nenhuma flag); portanto, ao fazer JR Z TECTV, a flag testada foi gerada pela instrução CP 'FF'. Uma vez achado o caractere, ele é colo-

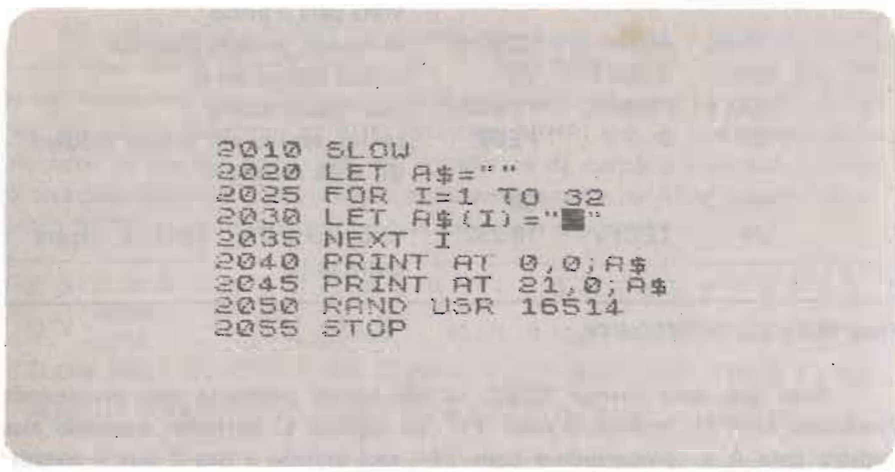
cado no acumulador e chama-se a sub-rotina PRINTC (vista no capítulo anterior) para colocá-lo na tela; se esse caractere for SPACE (código 'ØØ'), a sub-rotina retornará ao BASIC. Execute esse programa em modo SLOW e pressione qualquer tecla (menos SPACE porque senão o programa pára): conseguimos simular o efeito do INKEY$ em linguagem de máquina! Note a velocidade de resposta! Você pode imaginar como isso será útil para realizar jogos em tempo real (execute o programa em SLOW e depois em FAST...).

Façamos então uma pequena modificação para mover na direção vertical e horizontal um caractere na tela. Essa sub-rotina poderá ser muito útil para fazer jogos com movimento controlado pelas teclas 5, 6, 7 e 8 ou pelo joystick.

Procure colocar você mesmo os códigos que estão faltando em linguagem de máquina e, em particular, tome bastante cuidado nos cálculos para saltos relativos, principalmente os saltos para trás.

A lógica do programa que se segue é bastante simples (se baseia nos mesmos princípios do programa LABIRINTO) e acreditamos que seja relativamente fácil entendê-la sem maiores explicações: os comentários devem ser suficientes...

Execute o programa em SLOW e *não* desista até que você obtenha o resultado, ou seja, consiga mover o caractere X nas quatro direções. É necessário também um pequeno programa em BASIC para limitar a parte superior e inferior da tela com caracteres cujos códigos são maiores que 118. (Você saberia, estudando o programa em linguagem de máquina, explicar por quê?)



```
2010 SLOW
2020 LET A$=""
2025 FOR I=1 TO 32
2030 LET A$(I)="█"
2035 NEXT I
2040 PRINT AT 0,0;A$
2045 PRINT AT 21,0;A$
2050 RAND USR 16514
2055 STOP
```

Esse programa assume que a sub-rotina em linguagem de máquina esteja dentro de um REM na linha 1.


```
INKMAQ LD    HL,(D-FILE)               ; carrega HL com (D-FILE)
       LD    DE,115        '11.D115'
       ADD   HL,DE
       LD    A,X           '3E;X'      ; coloca X em (D-FILE) + 115
       LD    (HL),A
       PUSH  HL                        ; salva posição atual no STACK
TECTV  CALL  KDEC
       LD    B,H
       LD    C,L
       LD    A,L
       CP    'FF'
       JR Z, TECTV
       CALL  ACHR                      ; acha o código caractere corres-
                                         pondente
       LD    A,(HL)
       CP    5             'FE;5'      ; se for ⇦ , subtrai 1
       JR NZ,LOVE
       LD    DE,-1         '11FFFF'
       JR    PEACE
LOVE   CP    8                         ; se for ⇨ , soma 1
       JR NZ,LOVE1
       LD    DE,1
       JR    PEACE
LOVE1  CP    7                         ; se for ⇧ , subtrai 33
       JR NZ,LOVE
       LD    DE,-33        '11DFFF'
       JR    PEACE
LOVE2  CP    6                         ; se for ⇩ , soma 33
       JR NZ,LOVE3
       LD    DE,33
       JR    PEACE
LOVE3  CP    ␢             'FE;␢'      ; se for SPACE retorna ao
                                         BASIC
       JR Z, FIM           '2818'
       JR    TECTV
PEACE  POP   HL                        ; retira a posição do STACK
       LD    A,␢           '3E;␢'      ; coloca "branco" na posição
       LD    (HL),A
       ADD   HL,DE                     ; calcula nova posição
       LD    A,(HL)
```

```
        CP     N.L.      'FE,<>'   ; se a nova posição for NEW
                                     LINE ou qualquer caractere de
                                     código ≥ 118 não movimenta
                                     o X (LABEL AJS)
        JR NC,AJS
BOBO    LD     A,X                 ; caso contrário, movimenta o X
        LD     (HL),A
        PUSH HL                    ; salva no STACK a nova posição
        JR     TECTV               ; volta para testar outra tecla
AJS     SCF              '37'      ; ajusta o CARRY
        CCF              '3F'
        SBC    HL,DE     'ED52'    ; calcula a posição antiga e volta
                                     para BOBO
        JR     BOBO
FIM     POP    HL                  ; ajusta o SP
        RET                        ; volta ao BASIC
```

Prog. 1Ø.2 – INKMAQ (INKEY$ em linguagem de máquina).

Após traduzir todos os mnemônicos, reserve espaço com um REM e, utilizando HEXAMEM II, coloque o programa no computador (note que estamos usando b̸ para indicar "espaço em branco").

Digite então RUN 2Ø1Ø (veja a observação abaixo), aguarde alguns instantes (por quê?) e, finalmente, eis o caractere para ser controlado. Tente movimentá-lo de apenas *uma* posição...

Qual a tecla que pára a sub-rotina e volta ao BASIC? O que acontece se você pressionar qualquer tecla que não seja 5, 6, 7, 8 ou SPACE?

Você saberia explicar por que as instruções LD DE,–1 e LD DE,–33 têm como códigos hexadecimais '11FFFF' e '11DFFF', respectivamente? Compare a velocidade de "resposta" deste programa com o programa 9.1.

Observação: Antes de tentar executar o programa, verifique se sua instrução de RET está na memória 166Ø4. Em caso afirmativo, você já acertou pelo menos o número de bytes do programa. Basta verificar se os códigos estão certos. Sugerimos que você grave o programa assim como está, antes de executá-lo (juntamente com o HEXAMEM), pois, se caso ele não funcionar e ocorrerem "coisas estranhas", bastará recolocá-lo no computador e corrigi-lo com alguns PEEKs e POKEs, não necessitando reescrever todo o programa. De fato, para "varrer" a memória, basta fazer um programa bem simples:


```
3000 PRINT "MEMORIA INICIAL=?"
3005 INPUT M
3010 SLOW
3015 LET T=PEEK M
3020 GOSUB 1000
3022 SCROLL
3025 PRINT M,X$
3030 LET M=M+1
3035 GOTO 3015
```

Prog. 1Ø.3 – Programa para ler um trecho de memória.

Assim, supondo que o HEXAMEM II esteja no computador, basta fazer RUN 3ØØØ e colocar 16514 para a memória inicial e você poderá conferir os mnemônicos e saber *onde* fazer POKEs para modificar o programa.

Vamos explicar agora como, a partir do código do caractere, o computador consegue saber sua forma para, a seguir, escrevê-lo na tela. Para isto ele armazena em outra parte da memória ROM, que vai de '1EØØ' até '1FFF', uma tabela com o *modelo* de cada caractere, usando 8 bytes para cada caractere. Por exemplo, a letra *B* é armazenada da seguinte maneira:

```
Ø Ø Ø Ø Ø Ø Ø Ø
Ø 1 1 1 1 1 Ø Ø
Ø 1 Ø Ø Ø Ø 1 Ø
Ø 1 1 1 1 1 Ø Ø
Ø 1 Ø Ø Ø Ø 1 Ø
Ø 1 Ø Ø Ø Ø 1 Ø
Ø 1 1 1 1 1 Ø Ø
Ø Ø Ø Ø Ø Ø Ø Ø
```

Fig. 1Ø.2 – A letra *B* na memória ROM do TK.

Você consegue enxergar a letra B nos 8 bytes acima? Para colocá-la na tela, o computador associa, a cada bit em 1, um ponto *escuro*, fazendo assim os caracteres.

Fig. 1Ø.3 – Letra B.

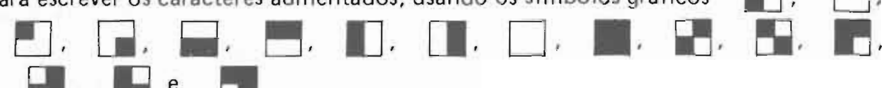
Assim, ao pressionar uma tecla, note o mecanismo utilizado para fazer aparecer o caractere na tela:

a) A tecla pressionada é decodificada pela sub-rotina KDEC, com resultados em HL.
b) As coordenadas que estão em HL são transferidas pelo programa interpretador para o par BC e é chamada a sub-rotina ACHR, que coloca em HL o endereço onde está o código do caractere.

c) De posse desse código, é chamada a sub-rotina PRINTC, que utiliza a tabela dos *modelos* dos caracteres (existente na ROM) para imprimir os mesmos na tela.

CARACTERES GIGANTES

Vamos apresentar a seguir dois programas bastante interessantes elaborados em 1981 por Toni Baker (Inglaterra). Eles utilizam os modelos que estão na ROM para escrever os caracteres aumentados, usando os símbolos gráficos

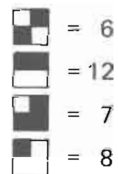


Vamos adotar uma regra de numeração para cada símbolo gráfico:

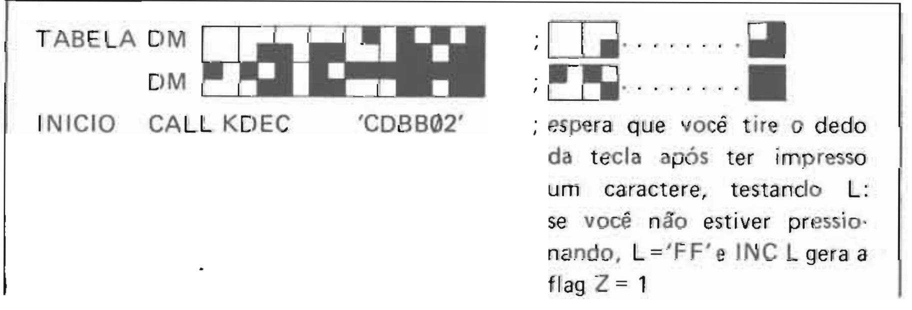
8	4
2	1

Fig. 1Ø.4 — Regra de numeração adotada para os caracteres gráficos.

Para saber o número de dado símbolo, basta verificar quais as porções em preto e *somar* os valores correspondentes. Assim, por exemplo, temos:



Assim, iremos fazer um programa que utiliza como *dados* os caracteres gráficos colocados em ordem crescente (segundo nossa regra) na memória:



```
TABELA DM [...]                  ; [...] ........ [...]
       DM [...]                  ; [...] ........ [...]
INICIO CALL KDEC      'CDBBØ2'   ; espera que você tire o dedo
                                   da tecla após ter impresso
                                   um caractere, testando L:
                                   se você não estiver pressio-
                                   nando, L='FF' e INC L gera a
                                   flag Z = 1
```



```
        INC L           '2C'
        JR NZ,INICIO    '2ØFA'
ESPERA  CALL KDEC       'CDBBØ2'      ; espera nova tecla, testando
                                        novamente L, e transferindo
                                        a saída para o par BC
        LD B,H          '44'
        LD C,L          '4D'
        LD D,C          '51'
        INC D           '14'          ; novamente, se não houver
                                        tecla pressionada, L = 'FF' e
                                        INC L gera a flag Z = 1 (se
                                        isto ocorrer, volta para ES-
                                        PERA)
        JR Z,ESPERA     '28F7'
        CALL ACHR       'CDBDØ7'      ; ao ser pressionada a tecla,
                                        acha o código do caractere
                                        e coloca em A
        LD A,(HL)       '7E'
        AND A           'A7'          ; "zera" a flag de CARRY sem
                                        alterar A
        RLA             '17'          ; três rotações para a esquerda
                                        multiplicam A por 8 (ver
                                        capítulo 6); se ao multiplicar
                                        por 4 houver CARRY, volta
                                        ao BASIC, pois não se trata
                                        de um caractere simples
        RLA             '17'
        RET C           'D8'
        RLA             '17'
        LD D,Ø          '16.Ø' ou '16ØØ' ; coloca CARRY gerado pelo
                                        último RLA no bit Ø de D,
                                        deixando os demais bits em
                                        zero
        RL D            'CB12'
        LD E,A          '5F'
        LD HL,TABROM    '21ØØ1E'      ; nesse ponto o par DE contém
                                        o número que, somado ao
                                        início da tabela (ROM), nos
                                        fornecerá o endereço do mol-
                                        de desejado
        ADD HL,DE       '19'          ; calcula endereço do início do
                                        molde
```

```
        LD C,4          'ØE.4'      ; C será contador para LOOP
LOOP    LD B,4          'Ø6.4'      ; B será contador para LOOP1
        LD D,(HL)       '56'        ; coloca os primeiros 2 bytes
                                      do molde no par DE
        INC HL          '23'
        LD E,(HL)       '5E'
        INC HL          '23'
        PUSH HL         'E5'        ; salva HL no STACK (pilha)
LOOP1   XOR A           'AF'        ; calcula qual símbolo gráfico
                                      será utilizado: zera A e
                                      CARRY
        RL D            'CB12'      ; rotaciona D duas vezes para a
                                      esquerda e coloca os CARRYs
                                      gerados nos bits Ø e 1 do acu-
                                      mulador
        RLA             '17'
        RL D            'CB12'
        RLA             '17'
        RL E            'CB13'      ; rotaciona E duas vezes para a
                                      esquerda e coloca os CARRYs
                                      gerados nos bits 2 e 3 do
                                      acumulador
        RLA             '17'
        RL E            'CB13'
        RLA             '17'
        LD HL,TABELA    '21.3ØØØØ'  ; soma o número obtido no
                                      acumulador (Ø a 15) com o
                                      início da tabela dos símbolos
                                      gráficos para obter o símbolo
        ADD A,L         '85'
        LD L,A          '6F'
        LD A,(HL)       '7E'        ; coloca o símbolo no acumu-
                                      lador
        EXX             'D9'        ; salva contadores (B e C)
        CALL PRINTC     'CDØ8Ø8'    ; coloca o símbolo na TV
        EXX             'D9'        ; recupera contadores
        DJNZ LOOP1      '1ØE6'      ; volta para completar a pri-
                                      meira linha do caractere
        LD A; N.L.      '3E; <>'    ; imprime um NEW LINE
                                      para mudar de linha
        EXX             'D9'
```



```
        CALL PRINTC     'CDØ8Ø8'
        EXX             'D9'
        POP HL          'E1'        ; retira HL do STACK
        DEC C           'ØD'
        JR NZ,LOOP      '2ØD4'      ; volta para imprimir a próxima
                                      linha do caractere
        JR INICIO       '18AF'
```

Prog. 1Ø.4 – Caracteres gigantes (primeira parte).

Coloque o programa a partir da memória 3ØØØØ, começando pelos símbolos gráficos precedidos por ponto e vírgula, e o execute digitando XX, seguido pelo endereço inicial, que é .3ØØ16 (por quê?), e, finalmente, XS. Tente agora apertar qualquer tecla que contenha um caractere simples (ou seja, sem ser STOP, LPRINT etc.) com ou sem SHIFT. O que acontece quando a tela acaba?

Como você deve ter notado no programa TECTV, a resposta em linguagem de máquina é excessivamente rápida, o que nos obriga a fazer uma parte do programa que espera você tirar o dedo da tecla após imprimir o caractere. Feito isso, analogamente ao TECTV, detectamos qual o caractere pressionado e multiplicamos o código por 8 para podermos achar o endereço inicial de seu molde na ROM (isso porque cada molde ocupa 8 bytes). Entretanto, se após multiplicar por 4 houver um CARRY, o programa voltará ao BASIC, pois não se trata de um caractere simples (verifique isso no manual do TK... basta multiplicar por 4 o código do primeiro caractere *não simples*, que é RND, e obteremos 4 x 64 = 256, que gera CARRY!). Assim, basta adicionar o número obtido pela multiplicação ao endereço inicial da tabela na ROM e começar a construção do caractere gigante.

Usamos dois contadores que vão até 4 (4 x 4 = 16) não até 8 (8 x 8 = 64), como seria esperado, pois construiremos quatro pontos por vez (de fato, cada símbolo gráfico corresponde em área a quatro pontos de PLOT). Para cada linha, colocamos os primeiros 2 bytes do molde no par DE e, ao entrar no LOOP1, para cada ciclo, 2 bits de D e 2 bits de E são colocados em A, gerando um número de Ø a 15 (que corresponde ao símbolo gráfico desejado), o qual é colocado na tela (tente fazer esse processo em câmara lenta para a letra B, cujo molde já foi apresentado). Este número é somado ao início da tabela de símbolos gráficos para nos fornecer em HL o endereço do símbolo correspondente, o qual é colocado no acumulador. Salvamos então os registros de B a L (EXX), colocamos o caractere na tela (CALL PRINTC) e recuperamos os registros. A instrução DJNZ LOOP1 testa a condição de fim de linha: quando B é zero, a linha termina e o programa prossegue; ao fim de cada linha é colocado um NEW LINE (por quê?) e decrementa-se C (contador de linhas); se as linhas terminarem, volta para o INICIO do programa para aguardar outro caractere.

Esse programa é bastante complicado e seria bom que você o entendesse em detalhes, simulando cada passo com a ajuda do apêndice 2 e dos capítulos anteriores. Feito isso, você poderá prosseguir.

Se você não possuir SLOW em seu computador, use o seguinte programa em BASIC:

```
10 RAND USR 30016
20 PAUSE 400
30 RUN
```

Se você não tiver expansão de memória, basta utilizar um valor menor para RAMTOP (mas tome cuidado com o início da sub-rotina pois ela não será mais 30016).

Após ter entendido o programa, você poderá melhorá-lo um pouco acrescentando algumas instruções:

TABELA	DM		; ....
	DM		; ....
BIGCHR	LD HL,(D-FILE)	'2A.16396'	; coloca a posição de PRINT no começo da tela
	INC HL	'23'	
	LD (DF-CC),HL	'22.16398'	
CURSOR	LD (HL); [>]	'36; [>] '	; coloca um cursor na tela
INICIO	CALL KDEC	'CDBB02'	; espera que você tire o dedo da tecla após ter impresso um caractere, testando L: se você não estiver pressionando, L = 'FF' e INC L gera a flag Z = 1
	INC L	'2C'	
	JR NZ,INICIO	'20FA'	
ESPERA	CALL KDEC	'CDBB02'	; espera nova tecla, testando novamente L, e transferindo a saída para o par BC
	LD B,H	'44'	
	LD C,L	'4D'	
	LD D,C	'51'	

	INC D	'14'	; novamente, se não houver tecla pressionada, L = 'FF' e INC L gera a flag Z = 1 (se isto ocorrer, volta para ESPERA)
	JR Z,ESPERA	'28F7'	
	CALL ACHR	'CDBD07'	; ao ser pressionada a tecla, acha o código do caractere e coloca em A
	LD A,(HL)	'7E'	
	AND A	'A7'	; "zera" a flag de CARRY sem alterar A
	RLA	'17'	; três rotações para a esquerda multiplicam A por 8 (ver capítulo 6); se ao multiplicar por 4 houver CARRY, volta ao BASIC, pois não se trata de um caractere simples
	RLA	'17'	
	RET C	'D8'	
	RLA	'17'	
	LD D,0	'16.0' ou '1600'	; coloca CARRY gerado pelo último RLA no bit 0 de D, deixando os demais bits em zero
	RL D	'CB12'	
	LD E,A	'5F'	
	LD HL,TABROM	'21001E'	; nesse ponto o par DE contém o número que, somado ao início da tabela (ROM), nos fornecerá o endereço do molde desejado
	ADD HL,DE	'19'	; calcula endereço do início do molde
	LD C,4	'0E.4'	; C será contador para LOOP
LOOP	LD B,4	'06.4'	; B será contador para LOOP1
	LD D,(HL)	'56'	; coloca os primeiros 2 bytes do molde no par DE
	INC HL	'23'	
	LD E,(HL)	'5E'	
	INC HL	'23'	

```
        PUSH HL         'E5'          ; salva HL no STACK (pilha)
LOOP1   XOR A           'AF'          ; calcula qual símbolo gráfico
                                        será utilizado: zera A e
                                        CARRY
        RL D            'CB12'        ; rotaciona D duas vezes para a
                                        esquerda e coloca os CARRYs
                                        gerados nos bits Ø e 1 do acu-
                                        mulador
        RLA             '17'
        RL D            'CB12'
        RLA             '17'
        RL E            'CB13'        ; rotaciona E duas vezes para a
                                        esquerda e coloca os CARRYs
                                        gerados nos bits 2 e 3 do
                                        acumulador
        RLA             '17'
        RL E            'CB13'
        RLA             '17'
        LD HL,TABELA    '21.3ØØØØ'    ; soma o número obtido no
                                        acumulador (Ø a 15) com o
                                        início da tabela dos símbolos
                                        gráficos para obter o símbolo
        ADD A,L         '85'
        LD L,A          '6F'
        LD A,(HL)       '7E'          ; coloca o símbolo no acumu-
                                        lador
        LD HL,(DF-CC)   '2A.16398'
        LD (HL),A       '77'          ; coloca o caractere na posição
                                        adequada
        INC HL          '23'
        LD (DF-CC),HL   '22.16398'    ; armazena a nova posição
                                        para PRINT
        DJNZ LOOP1      '1ØE3'
        PUSH DE         'D5'          ; salva DE
        LD DE,'1D'      '111DØØ'      ; ajusta posição do PRINT para
                                        a próxima linha
        ADD HL,DE       '19'
        LD (DF-CC),HL   '22.16398'
        POP DE          'D1'          ; recupera DE
        POP HL          'E1'          ; recupera HL
        DEC C           'ØD'
```



```
        JR NZ,LOOP      '2ØCF'        ; vai executar a próxima linha
        LD DE,'FF8Ø'    '118ØFF'
        LD HL,(DF-CC)   '2A.16398'    ; ajusta posição de PRINT para
                                        o próximo caractere
        ADD HL,DE       '19'
        LD (DF-CC),HL   '22.16398'
        LD A,(HL)       '7E'
        CP N.L.         'FE;<>'       ; verifica se foi atingido o
                                        fim da linha na tela
        JR NZ,CURSOR    '2Ø9B'        ; em caso negativo, volta para o
                                        início
        LD DE,'64'      '1164ØØ'      ; em caso positivo ajusta a posi-
                                        ção de PRINT para a próxima
                                        linha da tela
        ADD HL,DE       '19'
        LD (DF-CC),HL   '22.16398'
        INC HL          '23'
        LD DE,(VARS)    'ED5B.164ØØ'  ; verifica se foi atingido o fim da
                                        tela com a ajuda de VARS
        SBC HL,DE       'ED52'
        ADD HL,DE       '19'
        JR C,CURSOR     '388A'        ; em caso negativo, volta para
                                        o início
        RET             'C9'          ; em caso positivo, termina
                                        o programa
```

Prog. 10.5 — BIGCHR (BIG CHaRacter — caracteres gigantes completo).

Você terá de reescrever todo o programa novamente, a não ser que tenha resolvido os exercícios do capítulo 7 para inserir ou retirar blocos na memória. Tente então executar o programa. (Lembre-se: XX,.3ØØ16 e XS.)

Esse programa utiliza outra variável do programa interpretador, chamada DF-CC, na qual está a posição de PRINT no arquivo de imagem da tela (endereços 16398 e 16399). Iremos utilizar esses 2 bytes apenas para colocar as posições da tela onde vão ser impressos os símbolos gráficos. Na realidade, qualquer outra posição livre serviria, mas fizemos isso com o intuito de mostrar, *aproximadamente*, como o programa interpretador trabalha para colocar caracteres na tela utilizando D-FILE e DF-CC. A posição em DF-CC é ajustada para cada símbolo impresso do caractere gigante, para cada vez que um caractere for terminado e para cada vez que termina uma linha da tela (estude cuidadosamente esses ajustes e use a figura 9.2 para ajudá-lo).

De fato, inicialmente HL é carregado com o conteúdo de D-FILE e incrementado para indicar a primeira posição da tela, a qual é copiada em DF-CC. Colocamos então um cursor na tela. Segue uma parte idêntica ao programa 1Ø.4, que escolhe o símbolo gráfico a ser impresso, o qual é colocado na tela (LD (HL), A), na posição indicada por DF-CC. A sua posição é incrementada até terminar a linha, quando então a nova posição é calculada somando-se '1D' para obter o início da próxima linha (você saberia explicar por quê?). Ao terminar o caractere, o cursor e a nova posição devem ser novamente ajustados, o que é feito somando-se 'FF8Ø' (por quê?), e, ao chegar no fim da linha da tela (basta verificar se foi atingido um NEW LINE) a posição deve ser somada com '64' (por quê?).

Se você conseguir responder a todos esses "por quês" relativos aos ajustes na posição de impressão, além de conseguir entender claramente o programa, estará bem preparado para entender o programa LIFE no próximo capítulo.

Para testar o fim da tela, utilizamos o valor da variável VARS, que indica o início da região onde ficam as variáveis dos programas BASIC. Essa região fica logo após a tela de TV (ver capítulo 8). Tente explicar como funciona essa detecção de fim de tela. Estude com detalhes esse programa antes de prosseguir! Se você quiser transferi-lo para um REM, veja o capítulo 9.

Para finalizar o capítulo, aqui vai uma sugestão: se estiver com problemas de espaço na memória, além dos métodos sugeridos no capítulo 8 sobre como substituir números por STRINGs e, para a linguagem de máquina, como utilizar EXX em vez de 3 PUSHs ou 3 POPs (capítulo 7), você poderá substituir a sub-rotina PRINTC do capítulo anterior por uma parecida, chamada PRINTS que está na ROM a partir do endereço 'ØØ1Ø'. Essa sub-rotina tem duas grandes vantagens:

a) *Não* destrói os registros, não necessitando, portanto, de PUSHs, POPs ou EXX.
b) Pode ser chamada com uma instrução de RST em vez de CALL:

RST 16 ou RST '1Ø' (ver capítulo 7)

A instrução RST é mais rápida que a CALL e ocupa apenas um byte ao invés de três.

Resumo

Neste capítulo vimos três importantes sub-rotinas da ROM que servem, respectivamente, para detectar qual tecla foi pressionada (KDEC), apontar para a região da ROM onde estão os códigos dos caracteres (ACHR) e substituir a PRINTC vista no capítulo 9 com economia de memória (PRINTS). Vimos também em que região da ROM está a tabela com os modelos dos caracteres do TK.

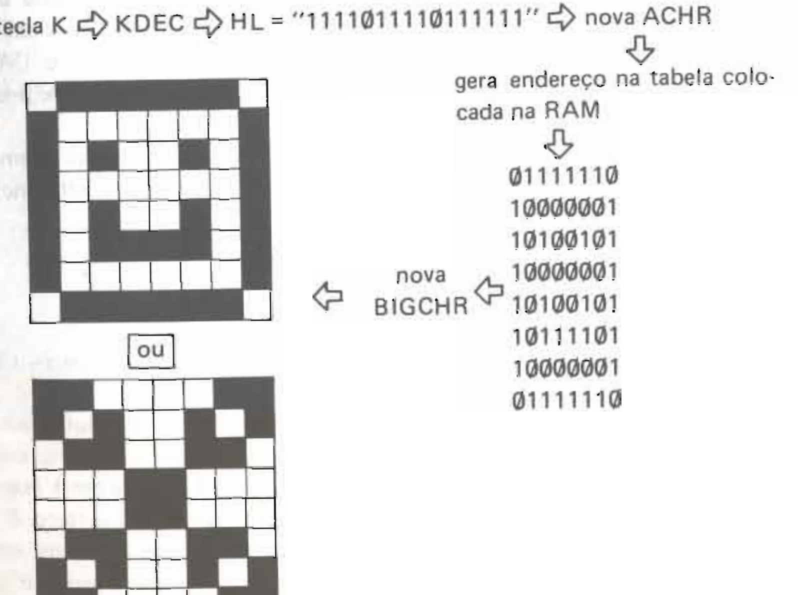
Foi explicado o método da decodificação do teclado e feitos dois programas: um para simular um INKEY$ em linguagem de máquina e outro para imprimir caracteres gigantes na tela. Esse último programa recorda muitos conceitos da

estrutura de programação em linguagem de máquina e seria aconselhável você estudá-lo antes de fazer os exercícios e prosseguir para o próximo capítulo.

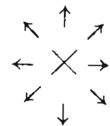
Exercícios

1. Modifique o programa BIGCHR para que ele imprima os caracteres gigantes em *vídeo-reverso*, ou, melhor ainda, faça com que você possa escolher durante o programa caracteres normais ou reversos.
2. Faça um programa que utilize a sub-rotina PRINTS para encher a tela com um caractere qualquer e, a seguir, faça um SCROLL da tela para a *esquerda* ou para a *direita*, mas *não* utilize o programa de SCROLL horizontal do capítulo 9. Como você deve estar lembrado, aquele SCROLL deslocava linha por linha da tela de uma posição. Refaça então o programa para que ele desloque coluna por coluna de uma posição.
3. Utilize a sub-rotina KDEC para detecção de teclas e faça uma sub-rotina que trabalhe de modo semelhante a ACHR mas que, em vez de fornecer como saída um endereço na ROM, forneça um endereço na RAM onde estarão os moldes de novos caracteres *criados por você*. Feito isso, adapte o programa BIGCHR para que ele desenhe esses caracteres para você.

Assim, utilizando a imaginação, você poderá *criar uma tabela* de caracteres bastante interessante e colocá-la na RAM com a ajuda do HEXAMEM II, sendo que a cada caractere estará associada uma tecla. Por exemplo:



4. Você saberia utilizar BIGCHR para imprimir seqüencialmente todos os caracteres cujos modelos estão na tabela da ROM? Imprima uma seqüência de caracteres, sempre checando o fim da tela.
5. Modifique o programa TECTV (1Ø.1) para que ele faça "correr" horizontalmente um caractere pela tela apagando a impressão anterior. Note que agora você não poderá mais utilizar a sub-rotina PRINTC.
6. Modifique o programa INKMAQ (1Ø.2) para que ele movimente o caractere para cima se qualquer tecla da primeira fileira for pressionada (de 1 a Ø, setores horizontais 3 e 4); para a esquerda se for pressionada qualquer tecla dos setores horizontais 1 e 2; para a direita se for pressionada qualquer tecla dos setores horizontais 5 e 6; e para baixo se qualquer tecla da última fileira for pressionada (de Z a SPACE, setores horizontais Ø e 7)
7. Implemente o programa anterior de maneira que o caractere possa "andar" também nas direções diagonais:



8. O que aconteceria no programa INKMAQ (1Ø.2) se não tivéssemos colocado as instruções SCF e CCF?
9. Por que, no programa BIGCHR (1Ø.5), ao se somar o número de Ø a 15 (obtido no LOOP) com o endereço da tabela de símbolos gráficos *não* é necessário fazer LD A,Ø/ADC A,H/LD H,A para "transmitir" o CARRY?
10. Como você faria para tornar *mais lenta* a velocidade de resposta do programa INKMAQ?
   Sugestão: Introduza nos lugares adequados alguns loops que não fazem nada!
11. Modifique o programa INKMAQ para que ele utilize toda a tela, incluindo as duas linhas de edição.

LIFE: o Computador Também é Artista

Para finalizar com chave de ouro este livro, apresentaremos um programa elaborado em 1970 por John Conway, da Universidade de Cambridge, Inglaterra.

O algoritmo utilizado gera ciclos de *vida* (nascimento/crescimento/morte) de uma colônia de células que vivem juntas e simetricamente... o efeito é realmente muito bonito.

O princípio de LIFE baseia-se num pequeno programa para gerar posições aleatórias e preencher aproximadamente um quarto de um quadrado de 16 x 16 posições (no nosso caso) com células. A partir daí, os ciclos são gerados obedecendo à seguinte regra: cada célula tem 8 possíveis células vizinhas.

1	2	3
4	●	5
6	7	8

Fig. 11.1 – Célula e suas vizinhas.

As células que tiverem apenas duas ou três vizinhas sobreviverão para o próximo ciclo e as demais morrerão. Uma nova célula nascerá em cada espaço vazio que tiver precisamente três células vizinhas. Note que as células das bordas terão 8 possíveis vizinhas, pois consideraremos esse quadrado de 16 x 16 como se fosse imaginariamente fechado em si mesmo; assim, por exemplo, a célula que estiver na borda superior terá como vizinhas células da borda inferior e assim por diante...

Fig. 11.2 – Célula [●] e suas vizinhas V.

Os cálculos para determinar as posições vizinhas serão feitos somando-se deslocamentos adequados à posição de cada célula.

Para conseguirmos o efeito desejado, usaremos um pequeno "truque": limitaremos *todas* as somas dentro de um byte, desprezando os CARRY que ocorrerem, e usaremos os seguintes deslocamentos para calcular as posições das células vizinhas:

'EF', 'Ø1', 'Ø1', '1Ø', '1Ø', 'FF', 'FF', 'FØ'

Por exemplo, comecemos com um caso simples: endereço inicial do quadrado (16 x 16) 'ØØ' e célula pesquisada, endereço '12':

'ØØ'	'Ø1' V	'Ø2' V	'Ø3' V		'ØF'
'1Ø'	'11' V	'12' [●]	'13' V		
'2Ø'	'21' V	'22' V	'23' V		

Fig. 11.3 – Cálculo das vizinhas da célula na posição '12'.

assim, as células vizinhas serão:

```
  '12'       '01'       '02'       '03'       '13'       '23'       '22'
+ 'EF'     + '01'     + '01'     + '10'     + '10'     + 'FF'     + 'FF'
 '101'      '02'       '03'       '13'       '23'      '122'      '121'
posição    posição    posição    posição    posição    posição    posição
  '01'       '02'       '03'       '13'       '23'       '22'       '21'

  '21'
+ 'F0'
 '111'
posição
  '11'
```

Verifique que as posições obtidas correspondem às vizinhas da célula (esse "truque" está relacionado com a convenção de números negativos; por exemplo, –1 é representado por 'FF').

Tente verificar isso para as figuras 11.2 e 11.4.

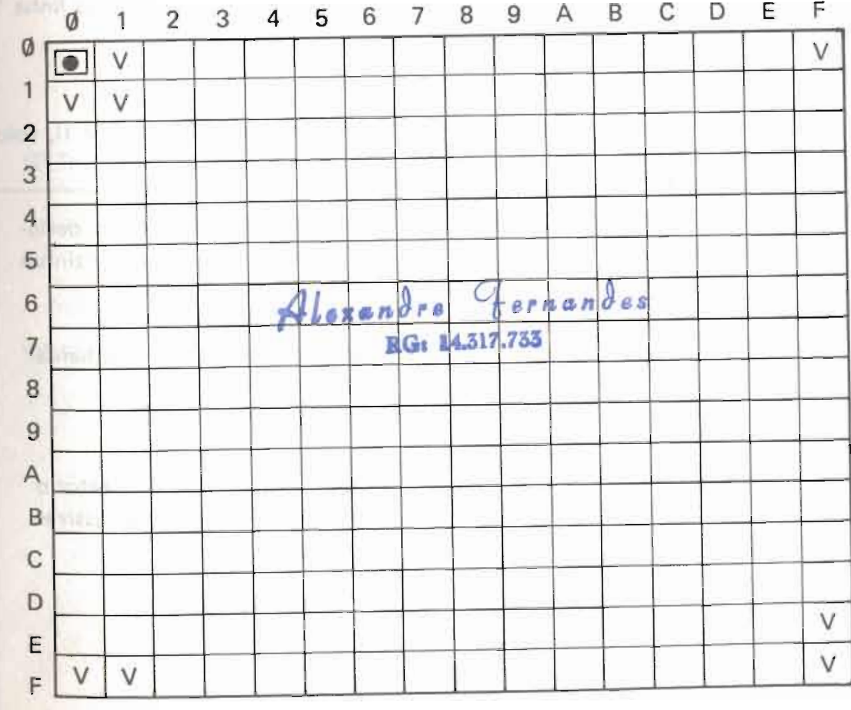


Fig. 11.4 – Célula na posição 'ØØ' e suas vizinhas.

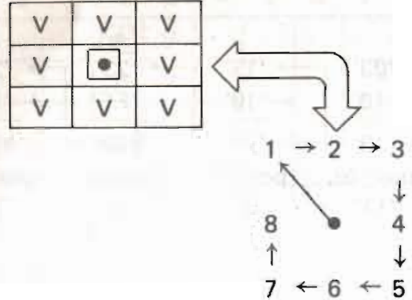


Fig. 11.5 – Seqüência de geração das células vizinhas.

Faça agora RAMTOP = 29952 ('75ØØ'):

```
POKE  16388,Ø
POKE  16389,117
NEW
```

Coloque então HEXAMEM II no computador e reserve, com um REM na linha 1, espaço para pelo menos 14Ø bytes:

```
1  REM  Ø123456789Ø1234 ...... 6789
```

Segue o programa em linguagem de máquina; usando HEXAMEM II, coloque-o a partir da memória 16514.

```
TABELA   DC  'EFØ1Ø11Ø1ØFFFFFØ'  ; dados que representam deslo-
                                   camentos para células vizinhas
ILIFE    LD C,16         'ØE.16'   ; C = contador de linhas
LOOPF    LD B,16         'Ø6.16'   ; B = contador de colunas
LOOP     LD HL,(SEED)    '2A.16434' ; carrega HL com "semente"
                                   gerada pelo RAND
         LD D,H          '54'      ; transfere HL para DE
         LD E,L          '5D'
         ADD HL,HL       '29'      ; gera um número aleatório
                                   através de somas sucessivas
                                   com realimentação
         ADD HL,HL       '29'
         ADD HL,DE       '19'
         ADD HL,HL       '29'
         ADD HL,HL       '29'
         ADD HL,HL       '29'
         ADD HL,DE       '19'
```



```
         LD (SEED),HL    '22.16434' ; coloca o número gerado na
                                    "semente"
         LD A,H          '7C'       ; baseado no byte mais signifi-
                                    cativo do número, decide se vai
                                    ou não "criar" uma célula
         CP 'C4'         'FEC4'
         JR C,CELULAN    '38Ø4'
         LD A,[o]        '3E;[o]'   ; coloca a célula no acumulador
         JR CARACT       '18Ø2'     (letra O em vídeo-reverso)
CELULAN  LD A,■          '3E;■'     ; coloca "vazio" no acumulador
CARACT   RST 16          'D7'       ; imprime o caractere
         DJNZ LOOP       '1ØE3'     ; volta para continuar a linha se
                                    B ≠ Ø
         LD A,N.L.       '3E;<>'    ; no fim da linha, imprime um
                                    NEW LINE
         RST 16          'D7'
         DEC C           'ØD'
         JR NZ,LOOPF     '2ØDB'     ; volta para fazer a próxima
                                    linha se C ≠ Ø
         RET             'C9'       ; volta ao BASIC após terminar
                                    a primeira geração
GERLIFE  LD B,Ø          'Ø6ØØ'     ; B = contador para posições das
                                    células (16 x 16 = 256)
         LD DE,PROV      '11.29952' ; carrega DE com endereço ini-
                                    cial para região de trabalho
         LD HL,(D-FILE)  '2A.16396' ; carrega HL com (D-FILE)
         PUSH HL         'E5'       ; coloca no STACK HL(D-FILE)
COPIA    LD A,(HL)       '7E'       ; copia caractere da tela no acu-
                                    mulador
         INC HL          '23'       ; aponta HL para o próximo ca-
                                    ractere
         CP '7F'         'FE7F'     ; se for NEW LINE ou espaço
                                    em branco, não copia o carac-
                                    tere na região de trabalho
         JR C,COPIA      '38FA'
         LD (DE),A       '12'       ; caso contrário, copia caractere
                                    na região de trabalho e aponta
                                    DE para próximo byte na re-
                                    gião de trabalho
         INC DE          '13'
```

	DJNZ COPIA	'10F6'	; ao copiar os 256 caracteres, prossegue
	LD DE,PROV	'11.29952'	; carrega DE novamente com início da região de trabalho
	PUSH DE	'D5'	; salva DE no STACK
PROXCEL	LD C,0	'0E00'	; C = contador para número de células vizinhas
	POP DE	'D1'	
	POP HL	'E1'	; recupera HL = (D-FILE)
PROC	LD A,(HL)	'7E'	; coloca caractere da tela em A
	CP '7F'	'FE7F'	; se for NEW LINE ou espaço em branco, faz HL apontar para o próximo caractere
	JR NC,OK	'3003'	
	INC HL	'23'	
	JR PROC	'18F8'	; volta para testar outro caractere
OK	PUSH HL	'E5'	; salva HL = posição da célula na tela
	EX DE,HL	'EB'	
	PUSH HL	'E5'	; salva DE = posição da célula na região de trabalho
	LD DE,TABELA	'11.16514'	; carrega DE com endereço inicial da tabela de deslocamentos
PROXDES	LD A,(DE)	'1A'	; carrega o deslocamento no acumulador
	CP '0E'	'FE0E'	; se for 0E, terminou a procura
	JR Z,FIMCONT	'280B'	
	INC DE	'13'	; aponta para o próximo deslocamento
	ADD A,L	'85'	; soma o valor do deslocamento com a posição atual da célula na área de trabalho, desprezando o CARRY
	LD L,A	'6F'	
	LD A,(HL)	'7E'	
	CP [o]	'FE; [o] '	; se houver uma célula, incrementa C (contador de células vizinhas); caso contrário, volta sem incrementar C para PROXDES
	JR NZ,PROXDES	'20F3'	

	INC C	'0C'	
	JR PROXDES	'18F0'	; volta para pesquisar mais uma vizinha
FIMCONT	POP HL	'E1'	; ao terminar, carrega posição da célula na área de trabalho em HL
	LD A,C	'79'	
	CP 2	'FE02'	; se a célula tem menos que duas vizinhas, ela morre
	JR C,CELNAO	'380F'	
	CP 4	'FE04'	; se a célula tem mais que quatro vizinhas, ela morre
	JR NC,CELNAO	'300B'	
	CP 3	'FE03'	; se a célula (ou espaço) tem três vizinhas, ela vive (ou é criada)
	JR Z,CELSIM	'2803'	
	LD A,(HL)	'7E'	; se não for nenhum destes casos, permanece como estava
	JR COLOCAR	'1806'	
CELSIM	LD A, [o]	'3E; [o] '	; coloca a célula no acumulador
	JR COLOCAR	'1802'	
CELNAO	LD A, ■	'3E; ■ '	; coloca "espaço" no acumulador
COLOCAR	EX (SP),HL	'E3'	; coloca em HL o topo da pilha (D-FILE + posição correspondente) *sem* alterar SP
	LD (HL),A	'77'	; coloca o caractere na tela
	INC HL	'23'	; aponta HL para a próxima posição
	EX (SP),HL	'E3'	; coloca HL no topo da pilha *sem* alterar SP e recupera a posição na região de trabalho
	INC HL	'23'	; aponta para a próxima posição
	PUSH HL	'E5'	; recoloca no topo da pilha
	LD A,L	'7D'	; se L atingir 0, isso significa que HL = 4400 (17152 + 256) e foram verificadas todas as células
	AND A	'A7'	
	JR NZ,PROXCEL	'20BD'	

POP HL	'E1'	; faz 2 POPs para colocar em SP seu valor inicial e retorna ao BASIC
POP HL	'E1'	
RET	'C9'	

Prog. 11.1 – Programa LIFE (16 X 16).

A seguir, acrescente o seguinte programa:

```
1995 SLOW
2000 RAND
2005 LET NADA=USR 16522
2010 LET NADA=USR 16562
2015 GOTO 2010
```

Prog. 11.2 – Programa comando para LIFE.

(Se você não tiver SLOW, basta dar uma pausa na linha 2012.)

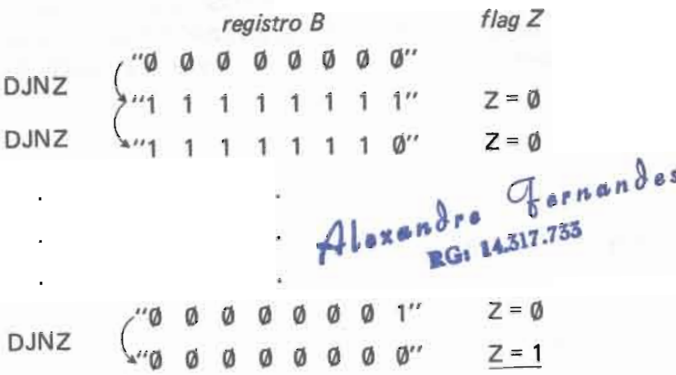
A seguir, digite RUN 1995. Eis o ciclo de vida de uma colônia de células. Pode ser que elas morram logo, ou que atinjam um equilíbrio estático ou dinâmico... tudo depende das posições aleatórias iniciais.

Esse programa é bastante complexo e utiliza muitos "truques" da linguagem de máquina, sendo que inclusive a nossa estrutura de colocação de comentários é precária para descrevê-lo. Uma estrutura mais complexa e precisa seria necessária mas não será aqui apresentada devido ao caráter introdutório do livro. Todavia, explicaremos os tópicos mais importantes, contando com a sua colaboração para o estudo de alguns detalhes.

Primeiramente, construímos uma tabela de 8 bytes, que correspondem aos deslocamentos necessários a serem dados a partir de uma dada posição de uma célula para achar suas vizinhas. Temos então a sub-rotina ILIFE, que gera o padrão inicial da colônia de células usando C e B como contadores para colocar as células nas devidas posições na tela. Um número aleatório é gerado carregando-se HL com a "semente" (SEED) gerada pela instrução RAND (que fica nos endereços 16434 e 16435) e através de somas sucessivas com realimentação; ou seja, somando-se sucessivamente HL e, algumas vezes, adicionando novamente o valor da semente, consegue-se um número aleatório (lembre-se que, no máximo, temos 16 bits e quando as somas dão resultados maiores que 65535 ('FFFF') os algarismos mais significativos são perdidos). (Tente simular no papel ou no próprio computador essa parte do programa.) A seguir, o número gerado é colocado na semente e, baseado no valor do registro H, decide-se se na tela irá ser colocada

ou não uma célula naquela posição. Isso é feito comparando-se o número gerado com 'C4' (por quê?) e utilizando-se a instrução RST 16 vista no capítulo 10 para colocar os caracteres na tela.

Finalmente é iniciada a sub-rotina GERLIFE, que gera um novo ciclo cada vez que ela é chamada. Aí já aparece um "truque": colocando 0 em B, poderemos contar 256 vezes, pois, a primeira vez em que é decrementado, ele resulta 255 sem colocar a flag Z em 1:



A seguir, outro "truque": é escolhida na RAM uma área de trabalho cujo endereço inicial é 29952 ou '7500' em hexadecimal; ora, após completar 256 posições ('100') teremos como final '7500' + '100' = '7600'... Assim, no final da sub-rotina, basta verificar apenas se o byte *menos* significativo atingiu 00, *não* sendo necessário comparar o byte mais significativo.

Segue-se o LABEL COPIA, que indica o início de uma transferência do conteúdo da tela (*menos* os NEW LINEs e espaços em branco), correspondente à nossa colônia, para uma região provisória na RAM, onde estará o modelo intacto da geração atual para ser construída a próxima geração. Essa região provisória é a área de trabalho mencionada anteriormente. (Essa parte do programa usa o registro B como contador para as 256 posições.) Antes do LABEL COPIA tivemos um PUSH HL, que salvou no STACK o endereço do início da tela, e, após o loop de cópia, tivemos um PUSH DE que colocou no STACK o endereço inicial da região de trabalho:

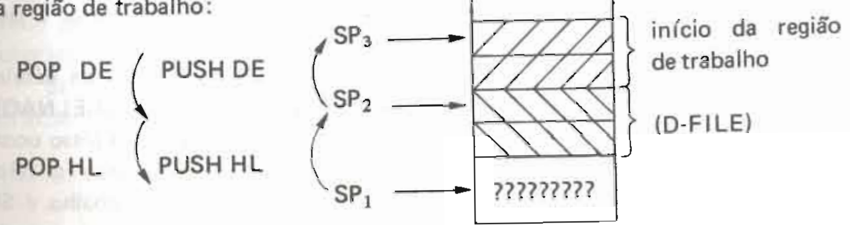


Fig. 11.6 – Visualização do STACK na memória.

Começa então uma contagem do número de células vizinhas para cada célula, onde o registro C é o contador. Os POPs carregam DE com o início da região de trabalho e HL com o início da tela, fazendo SP voltar a indicar SP_1 (figura 11.6); não se preocupe com estes POPs por enquanto! Eles serão facilmente entendíveis quando você repetir o raciocínio para a próxima célula. O caractere da tela é colocado em A e, se for um NEW LINE ou espaço em branco, incrementa-se HL para indicar a próxima posição na tela e retorna-se para PROC, até encontrar uma célula ou "espaço". O PUSH HL salva essa posição no STACK e, após a troca de DE por HL, um novo PUSH HL salva novamente DE (início da região de trabalho que corresponde à primeira célula) no STACK.

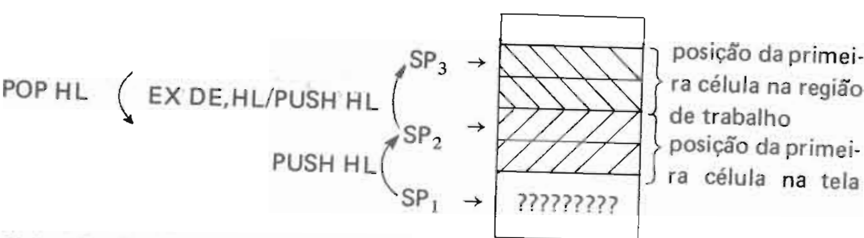


Fig. 11.7 – Visualização do STACK na memória.

Nesse ponto, HL contém o endereço da primeira célula na área de trabalho e DE é carregado com o endereço inicial da tabela que contém os deslocamentos necessários para calcular as posições das células vizinhas. O deslocamento é colocado no acumulador e, se este for 'ØE', isso significa que terminamos a tabela e estamos atingindo o programa (de fato o código de LD C,dado é 'ØE': veja o início do programa) e que, portanto, terminamos a contagem das células vizinhas e podemos pular para o label FIMCONT. Caso contrário, já incrementa DE para indicar a próxima posição na tabela de deslocamentos e soma-se, desprezando o CARRY, o deslocamento na posição atual da célula na área de trabalho, pois é lá que está apontando HL (você saberia explicar por quê?), achando-se assim a posição vizinha. Testa-se se há uma célula na posição e, em caso afirmativo, incrementa-se C e volta-se para procurar a próxima vizinha; caso contrário, C não é incrementado.

Chegamos então a FIMCONT... Ora, basta verificar o número de células vizinhas: se for menor que dois ou maior que quatro, a célula morre (label CELNAO), se for exatamente três, uma célula é criada (label CELSIM) e, se nada disso ocorrer, a célula permanece como estava. Nesse último caso, o POP HL, que foi feito em FIMCONT, recolocou em HL a posição da célula na área de trabalho e SP passou a indicar SP_2 (figura 11.7). Assim, ao fazer LD A,(HL) coloca-se o que havia nessa região no acumulador sem alterar nada...

Após alterar ou não a célula conforme o número de vizinhas, a instrução EX (SP),HL troca HL com o conteúdo do topo do STACK sem alterar SP (que continua a indicar SP_2). Desse modo, em HL temos a posição da célula na tela e no STACK, a posição da célula na região de trabalho (figura 11.7). Coloca-se então a célula (ou espaço) na tela, incrementa-se a posição da célula na tela, colocando-a no STACK (EX (SP),HL) e, a seguir, incrementa-se a posição na região de trabalho e coloca-se essa posição no STACK, usando PUSH HL.

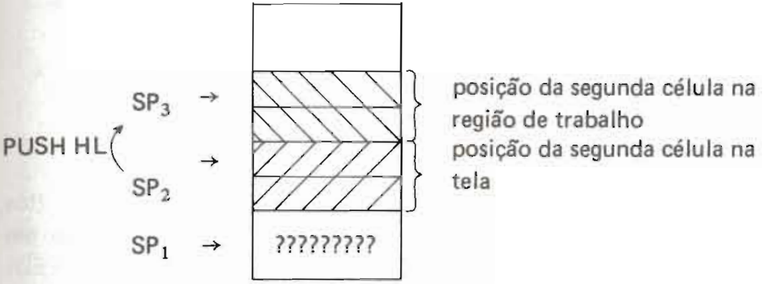


Fig. 11.8 – Visualização do STACK na memória.

Note que agora, no STACK (figura 11.8), temos uma situação para a segunda célula idêntica à da figura 11.7 para a primeira célula; basta checar se as células terminaram e voltar para a próxima célula (PROXCEL), onde agora fica claro o porquê dos POPs. Para testar se as células terminaram, usamos o "truque" mencionado no início sobre a escolha do endereço para a região de trabalho e as flags são geradas fazendo-se AND A. (Lembre-se que é necessário colocar o registro L em A antes disso, pois em HL está agora a posição da próxima célula na área de trabalho.)

Ao terminar todas as células, SP estará indicando a posição SP_3 (figura 11.8) sendo então necessários 2 POPs para fazê-lo voltar à posição original SP_1, onde está o endereço de retorno da sub-rotina para o BASIC (você saberia explicar por quê?).

Nesse programa foi utilizada uma área de trabalho provisória na RAM para garantir que cada nova geração fosse baseada totalmente na geração antiga *inalterada* pois, caso contrário, cada alteração surgida nas primeiras posições iria influenciar as próximas, "quebrando" assim o conceito de colônia de células.

Se você achar que o ritmo dos ciclos está muito rápido, pode introduzir um atraso no próprio programa em BASIC:

```
1997 LET C=0
2012 LET C=C+1
2014 PRINT AT 19,7;C
```

Baseado no que foi visto, você saberia explicar por que tivemos de fazer RAMTOP = 29952 ('75ØØ') apesar do programa estar num REM?

Este foi LIFE...LIFE é Vida... Vida é esperança, é "luta" e maravilhas para atingir o ideal do Universo... harmonia, paz e amor; esta é a resposta...

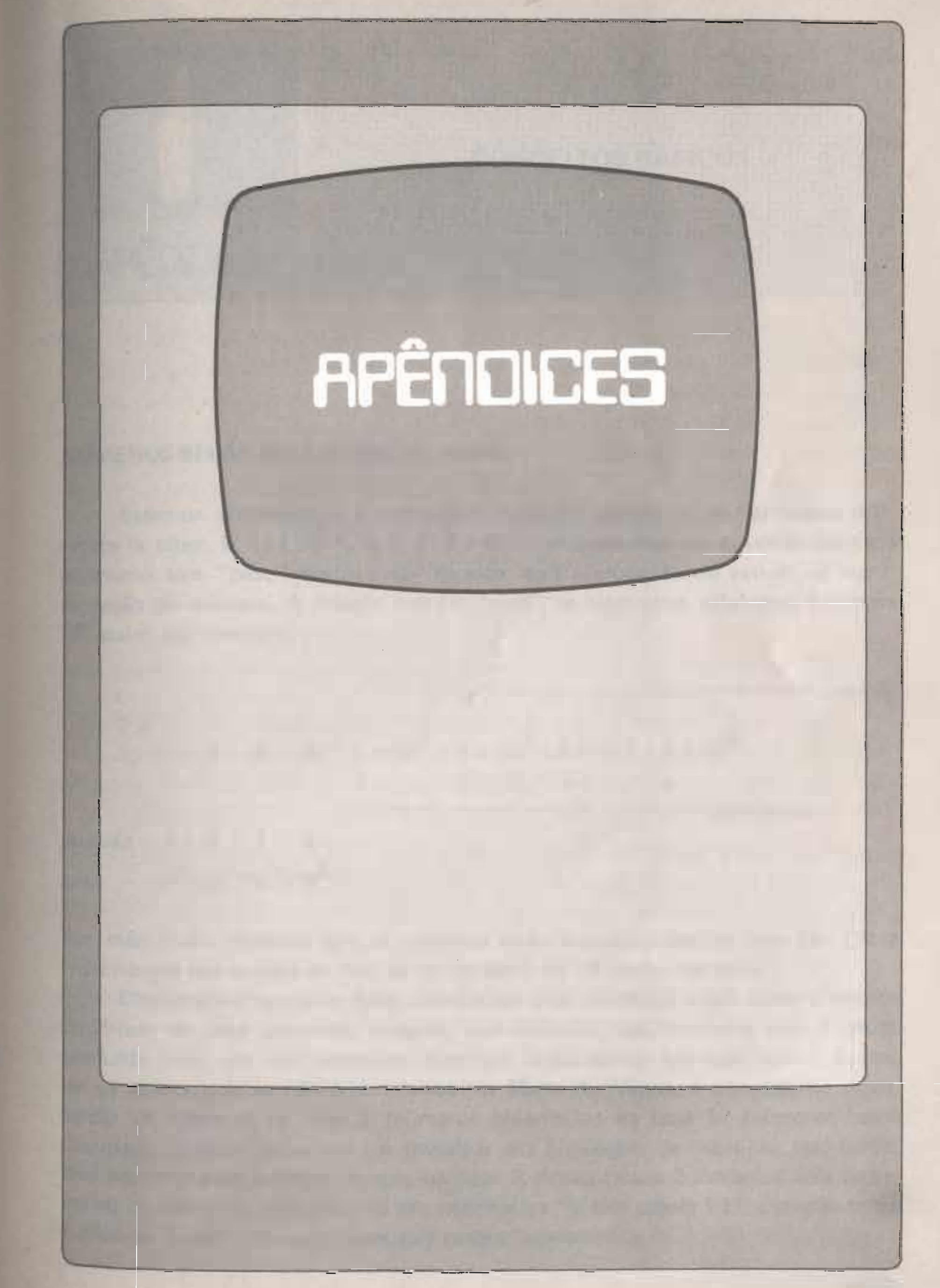
... Strawberry Fields Forever ...

Resumo

Em poucas palavras, vimos o programa LIFE, o qual apresentou muitos "truques" e conceitos, recordando firmemente a estrutura de programação em linguagem de máquina. Vimos também onde é colocada a "semente" (SEED) gerada pelo RAND (memória 16434 e 16435).

Exercícios

1. Você saberia explicar com detalhes o uso das instruções CP '7F'/JR C,COPIA e CP '7F'/JR NC,OK no programa LIFE?
2. Faça um programa LIFE para gerar uma colônia de 24 por 24 células ou, até, 24 por 32... Cuidado agora, pois alguns "truques" do LIFE original deixarão de funcionar pois 24 x 24 > 256!
3. Complemente o programa HEXAMEM II para ele aceite números negativos (o que será muito útil para as instruções JR e DJNZ) e associe-o aos programas feitos nos exercícios do capítulo 7 para que o HEXAMEM II permita a retirada e inserção de blocos de programa em linguagem de máquina.
4. Faça um programa que, se colocado no fim de uma sub-rotina em linguagem de máquina, coloque em uma dada região de memória o conteúdo de *todos* os registros internos do Z8Ø (incluindo o F, SP, IX, IY, A', B', C', D', E', H', L' e F') para que, utilizando PEEK, você possa analisá-los.
5. Como você faria para aumentar (ou diminuir) a probabilidade de gerar células da sub-rotina ILIFE?
6. Se não tivéssemos usado a "região de trabalho" no programa LIFE e nos baseássemos apenas na tela, os deslocamentos utilizados na TABELA seriam os mesmos? Por quê?



CONCEITOS BÁSICOS

NÚMEROS BINÁRIOS E HEXADECIMAIS

Estamos acostumados a representar números através de *dez* símbolos diferentes (a saber: Ø, 1, 2, 3, 4, 5, 6, 7, 8 e 9), chamados algarismos, sendo que cada algarismo tem "peso" tanto maior quanto mais à esquerda ele estiver na representação do número. A relação entre o "peso" de algarismos adjacentes é sempre 1Ø; assim, por exemplo:

posição →

$$1 \quad 9 \quad 8 \quad 3 = 1 \times 10^3 + 9 \times 10^2 + 8 \times 10^1 + 3 \times 10^0$$

→ *algarismos*

posição – 3 / 2 / 1 / Ø
peso – $1Ø^3/1Ø^2/1Ø^1/1Ø^Ø$

Por essa razão, dizemos que os números estão representados na *base* 1Ø. Diz a História que isso se deve ao fato do ser humano ter 1Ø dedos nas mãos...

Obviamente, qualquer base poderia ter sido escolhida e quaisquer símbolos poderiam ter sido adotados; imagine, por exemplo, um marciano com 8 dedos em cada mão., ou um venusiano com um dedo apenas em cada mão... Apesar de os computadores não serem feitos em Marte ou Vênus, é conveniente representar os números na base 2 (números binários) e na base 16 (números hexadecimais) quando pensamos em trabalhar em linguagem de máquina, por razões que veremos mais adiante. Assim, na base 2 necessitamos 2 símbolos para representar os números, e na base 16 são necessários 16 (ver tabela 1.1); a relação entre o peso de 2 algarismos adjacentes será respectivamente *2* e *16*.

BASE		
2 ou B	10 ou D	16 ou H
Ø	Ø	Ø
1	1	1
	2	2
	3	3
	4	4
	5	5
	6	6
	7	7
	8	8
	9	9
		A
		B
		C
		D
		E
		F

Tab. ap. 1.1 – Símbolos usados para representar números binários (base 2), decimais (base 1Ø) e hexadecimais (base 16).

Note que na base 16 somos obrigados a utilizar letras para os últimos 6 símbolos, pois nós, seres humanos, temos apenas 1Ø símbolos numéricos (Ø, 1, 2, ..., 9). Assim, nos números hexadecimais, as letras A, B, C, D, E e F representam, com um único símbolo, os números 1Ø, 11, 12, 13, 14 e 15 da base 1Ø!

Observação: Para evitar confusão, colocaremos os números entre parenteses, com a base utilizada em baixo, à direita:

$(\text{número})_{\text{base}}$

Por exemplo, vamos escrever dois números na base 2 (ou B) e 2 números na base 16 (ou H) e suas respectivas representações na base 1Ø (ou D):

$$(1\ 1\ \emptyset\ 1)_2 = 1 \times 2^3 + 1 \times 2^2 + \emptyset \times 2^1 + 1 \times 2^\emptyset = (13)_{1\emptyset}$$

posição: 3 / 2 / 1 / Ø

peso: $2^3/2^2/2^1/2^\emptyset$

$$(1\ \emptyset\ \emptyset\ \emptyset\ 1\ \emptyset\ 1\ \emptyset) = 1 \times 2^7 + \emptyset \times 2^6 + \emptyset \times 2^5 + \emptyset \times 2^4 + 1 \times 2^3 + \emptyset \times 2^2 + 1 \times 2^1 + \emptyset \times 2^\emptyset = (138)_D$$

posição: 7 / 6 / 5 / 4 / 3 / 2 / 1 / Ø

peso: $2^7/2^6/2^5/2^4/2^3/2^2/2^1/2^\emptyset$

$$(1\ 3\ 2)_{16} = 1 \times 16^2 + 3 \times 16^1 + 2 \times 16^\emptyset = (3\emptyset6)_{1\emptyset}$$

posição: 2 / 1 / Ø

peso: $16^2/16^1/16^\emptyset$

$$(A\ 7)_H = \underset{(A)}{1\emptyset} \times 16^1 + 7 \times 16^\emptyset = (167)_D$$

posição: 1 / Ø

peso: $16^1/16^\emptyset$

Note que no último exemplo o símbolo A foi substituído pelo número 1Ø para que pudéssemos fazer a conta de modo convencional e obter o número $(A7)_H$ na base 1Ø. O mesmo ocorre para B, C, D, E e F, que foram usados na base 16 para poder representar, com um único símbolo, os números de 11 a 15 (que são representados com dois símbolos na base 1Ø).

Para melhor compreensão vamos completar a tabela apêndice 1.1 usando os símbolos da base 2 e base 1Ø para representar todos os símbolos da base 16.

BASE		
2 ou B	10 ou D	11 ou H
Ø	Ø	Ø
1	1	1
1Ø	2	2
11	3	3
1ØØ	4	4
1Ø1	5	5
11Ø	6	6
111	7	7
1ØØØ	8	8
1ØØ1	9	9
1Ø1Ø	1Ø	A
1Ø11	11	B
11ØØ	12	C
11Ø1	13	D
111Ø	14	E
1111	15	F

Tab. ap.1.2 – Símbolos da base 16 e sua representação em binário e decimal.

Assim, vimos como converter números representados na base 2 e na base 16 para a base 1Ø. Para passar da base 2 para 16, ou vice-versa, o procedimento é bem simples pois $16 = 2^4$.

Desse modo, para obter a representação em hexadecimal, de um número na base 2, basta agrupar os algarismos em grupos de 4, a partir da direita, e substituí-los pelo correspondente algarismo na base 16. (Você saberia explicar por quê?)

$(\underbrace{1Ø1}_{5}\ \underbrace{ØØ1Ø}_{2}\ \underbrace{Ø11Ø}_{6}\ \underbrace{11ØØ}_{C})_B = (5\,2\,6\,C)_H$

$(\underbrace{11}_{3}\ \underbrace{11Ø1}_{D}\ \underbrace{Ø111}_{7}\ \underbrace{1111}_{F})_2 = (3D7F)_{16}$

(Ver tabela ap.1.2.)

Dado um número na base 16, para obter sua representação em binário basta substituir cada algarismo pela sua correspondente representação na base 2. Note que são *sempre* necessários 4 algarismos de base 2 para cada símbolo da base 16, sendo que os espaços devem ser preenchidos com zeros à esquerda:

$(3BF)_{16} = (\underbrace{11}_{3}\ \underbrace{1Ø11}_{B}\ \underbrace{1111}_{F})_2$

$(A482)_H = (\underbrace{1Ø1Ø}_{A}\ \underbrace{Ø1ØØ}_{4}\ \underbrace{1ØØØ}_{8}\ \underbrace{ØØ1Ø}_{2})_B$

Para completar o ciclo das transformações de bases, resta apenas aprender a passar números na base 1Ø para as bases 2 e 16. Via de regra, o seguinte algoritmo pode ser utilizado para passar números da base 1Ø para qualquer base X (tente explicar o porquê):

a) Divida o número dado por X, anote o resto e o resultado.
b) Divida o resultado por X, anote o resto e o novo resultado.
c) Repita a operação *b* até o resultado obtido ser zero (obviamente, se o resultado do passo *a* for zero, não serão necessários os passos *b* e *c*).
d) Copie os restos seqüencialmente, começando pelo último.

Exemplo

Vamos passar alguns números da base 1Ø para a base 2:

$(35)_{1Ø} = (?)_2$

```
35 /2
 1  17/2
     1  8/2
        Ø  4/2
           Ø  2/2
              Ø  1/2
                 1  Ø
```

$\therefore (35)_{1Ø} = (1Ø\ ØØ11)_2$

$(92)_D = (?)_B$

```
92 /2
 Ø  46/2
     Ø  23/2
         1  11/2
             1  5/2
                1  2/2
                   Ø  1/2
                      1  Ø
```

$\therefore (92)_D = (1Ø1\ 11ØØ)_B$

e da base 1Ø para a base 16:

$(88)_{1Ø} = (?)_{16}$

```
88 /16
 8   5/16
     5  Ø
```

$\therefore (88)_{1Ø} = (58)_{16}$

$(216)_D = (?)_H$

```
216 /16
 56  13/16
  8  13  Ø
```

$\therefore (216)_D = (D8)_H$ (lembre-se que $(13)_{1Ø} = (D)_{16}$ — ver tabela ap. 1.1)

Observação: Note a conveniência de se agrupar os algarismos dos números binários de 4 em 4 para facilitar a compreensão, assim como fazemos para os números decimais (de 3 em 3).

Exercícios

1. Efetue as seguintes conversões:

a) $(1Ø\ 11Ø1)_B = (?)_D$

b) $(1Ø\ 1Ø)_{1Ø} = (?)_{16}$

c) $(A2F)_H = (?)_B$

d) $(38Ø)_{16} = (?)_{1Ø}$

e) $(1Ø1\ ØØ1Ø\ 1Ø11)_2 = (?)_{16}$

f) $(111\ Ø11Ø)_2 = (?)_{1Ø}$

g) $(A1ØF)_H = (?)_D$

h) $(1Ø\ 111Ø\ Ø11Ø\ 11ØØ)_B = (?)_H$

i) $(ACF2)_{16} = (?)_2$

j) $(127)_{1Ø} = (?)_2$

k) $(999)_D = (?)_H$

l) $(11Ø)_D = (?)_B$

Sugestão para uma tarde de verão: Algum tempo atrás era bastante usada também a base 8 (números octais); tente imaginar números representados nessa base e como transformá-los para as bases que aprendemos hoje e vice-versa (note que $8 = 2^3$); a seguir, faça o mesmo para uma base que não seja potência de 2. (ex.5).

Sugestão para cinco tardes de verão: Experimente imaginar números na base 12 e fazer contas com eles: soma, subtração, multiplicação, divisão...
Verifique que certas contas de divisão que dão dízima na base 1Ø não dão dízima na base 12; ex.: 4/3 = 1,333... (base 1Ø)
4/3 = 1,4 (base 12)

BIT E BYTE: CIRCUITOS ELETRÔNICOS DE DOIS ESTADOS

A evolução da eletrônica tornou possível a elaboração de circuitos extremamente pequenos capazes de assumir dois estados estáveis, sinalizados pela presença ou não de tensão elétrica. Para efeito de representação, esses estados são simbolizados, respectivamente, por 1 e Ø, razão pela qual torna-se necessário representar os números na base 2 se quisermos interpretar como números os estados dos circuitos. Os computadores são constituídos em sua grande parte por *milhares* desses circuitos inteligentemente ligados.

Na linguagem "usual", cada algarismo Ø ou 1 é chamado *bit* e a um conjunto de 8 bits chamamos de *byte*. Os números hexadecimais foram introduzidos para facilitar a escrita e compreensão de números binários, substituindo longas cadeias de zeros e uns, pelos correspondentes símbolos hexadecimais, reduzindo assim quatro vezes seu tamanho e facilitando o entendimento do ponto de vista humano.

Assim, se tivermos oito circuitos iguais ao mencionado anteriormente, podemos representar 256 números (2^8), de Ø a 255, correspondendo a todas as combinações possíveis de Øs e 1s:

"ØØØØ ØØØØ"	Ø	'Ø'
"ØØØØ ØØØ1"	1	'1'
"ØØØØ ØØ1Ø"	2	'2'
"ØØØØ ØØ11"	3	'3'
.	.	.
.	.	.
.	.	.
"1111 11ØØ"	252	'FC'
"1111 11Ø1"	253	'FD'
"1111 111Ø"	254	'FE'
"1111 1111"	255	'FF'
(binários)	(decimais)	(hexadecimais)

Assim, utilizando 1 byte (8 bits) podemos representar os números inteiros positivos de Ø a 255.

Se usássemos 16 bits (ou seja, 2 bytes), poderíamos representar 2^{16} números, a saber: de Ø a 65535. Essa regra é análoga à base 1Ø (de fato, com 2 (dois) algarismos poderemos representar $1Ø^2$ números (Ø a 99)).

Note que introduzimos aqui uma nova simbologia para diferenciar as bases nas quais estão representados os números: os binários entre aspas ("), os hexadecimais entre apóstrofes (') e os decimais *sem "nada"*; essa simbologia é a que foi adotada para todo o livro.

Como veremos no decorrer do livro, serão muito interessantes para nós os números de 1 byte (8 bits) e 2 bytes (16 bits). Assim como nos números decimais, onde existe o conceito de "algarismo mais significativo", podemos falar de "byte mais significativo" para números com mais de um byte. Por exemplo, o número hexadecimal:

```
          1 byte        1 byte
        ┌─────────┐   ┌─────────┐
'1A2C' = "ØØØ1 1Ø1Ø   ØØ1Ø 11ØØ"
          └──┘ └──┘   └──┘ └──┘
           1    A      2    C
```

têm como byte mais significativo '1A' e como byte menos significativo '2C'. Esse número poderia ser assim representado em decimal:

$$'1A2C' = 1 \times 16^3 + 1Ø \times 16^2 + 2 \times 16^1 + 12 \times 16^Ø = 67ØØ$$
$$\qquad\qquad (A) \qquad\qquad\qquad (C)$$

Poderíamos, porém, calculá-lo de outra forma:

byte mais significativo: $'1A' = 1 \times 16^1 + 1Ø \times 16^Ø = 26$

byte menos significativo: $'2C' = 2 \times 16^1 + 12 \times 16^Ø = 44$

e calcular:

(byte mais significativo) $\times 256^1$ + (byte menos significativo) $\times 256^Ø$ =

$= 26 \times 256^1 + 44 \times 256^Ø = 67ØØ$

como se estivéssemos usando uma base fictícia de 256 ($= 16^2$).

Para entender melhor, suponha, por exemplo, o número 1Ø24 em decimal; ele equivale a:

$1 \times 1Ø^3 + Ø \times 1Ø^2 + 2 \times 1Ø^1 + 4 \times 1Ø^Ø$

Se dividirmos o número ao meio (1Ø ¦ 24) poderemos usar uma base fictícia de $1Ø^2 = 1ØØ$; de fato:

$1Ø \times 1ØØ^1 + 24 \times 1ØØ^Ø = 1Ø24$

Não se trata de coincidência! É lógica!

Medite um pouco sobre esses conceitos. Eles serão úteis e necessários para entender a linguagem de máquina...

Para facilitar as coisas, apresentamos aqui uma tabela dos primeiros 256 números hexadecimais e decimais para conversão, ou seja, todos os números possíveis de serem representados com 1 byte e que, portanto, têm dois dígitos hexadecimais:

	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø	Ø	1	2	3	4	5	6	7	8	9	1Ø	11	12	13	14	15
1	16	17	18	19	2Ø	21	22	23	24	25	26	27	28	29	3Ø	31
2	32	33	34	35	36	37	38	39	4Ø	41	42	43	44	45	46	47
3	48	49	5Ø	51	52	53	54	55	56	57	58	59	6Ø	61	62	63
4	64	65	66	67	68	69	7Ø	71	72	73	74	75	76	77	78	79
5	8Ø	81	82	83	84	85	86	87	88	89	9Ø	91	92	93	94	95
6	96	97	98	99	1ØØ	1Ø1	1Ø2	1Ø3	1Ø4	1Ø5	1Ø6	1Ø7	1Ø8	1Ø9	11Ø	111
7	112	113	114	115	116	117	118	119	12Ø	121	122	123	124	125	126	127
8	128	129	13Ø	131	132	133	134	135	136	137	138	139	14Ø	141	142	143
9	144	145	146	147	148	149	15Ø	151	152	153	154	155	156	157	158	159
A	16Ø	161	162	163	164	165	166	167	168	169	17Ø	171	172	173	174	175
B	176	177	178	179	18Ø	181	182	183	184	185	186	187	188	189	19Ø	191
C	192	193	194	195	196	197	198	199	2ØØ	2Ø1	2Ø2	2Ø3	2Ø4	2Ø5	2Ø6	2Ø7
D	2Ø8	2Ø9	21Ø	211	212	213	214	215	216	217	218	219	22Ø	221	222	223
E	224	225	226	227	228	229	23Ø	231	232	233	234	235	236	237	238	239
F	24Ø	241	242	243	244	245	246	247	248	249	25Ø	251	252	253	254	255

Tab. ap. 1.3 – Tabela de conversão decimal/hexadecimal e vice-versa.

Assim, por exemplo, na linha 5, coluna A, teremos:

'5A' = 9Ø (de fato $5 \times 16^1 + 1Ø \times 16^Ø = 9Ø$)

e 247 corresponde à linha F, coluna 7, ou seja:

247 = 'F7'

O QUE É POKE – O QUE É PEEK

Como veremos, a memória do TK pode, com expansão de 64 K, ter 65536 *casinhas* (posições), numeradas de Ø a 65535, cada uma capaz de armazenar 1 byte, ou seja, um número compreendido entre Ø e 255 ('ØØ' e 'FF'). A instrução POKE coloca um número (de Ø a 255) na memória desejada; por exemplo:

```
POKE 3ØØØØ,135
```

coloca o número 135 na memória 3ØØØØ. A função PEEK nos "conta" qual o número que está em dada memória; por exemplo:

```
PRINT PEEK 3ØØØØ
```

escreve na tela o que está na memória 3ØØØØ. Lembre-se, os endereços só vão de Ø a 65535 e o número máximo que cabe em uma memória é 255. Assim, para colocar na memória um número maior que 255 devemos "quebrá-lo" em várias partes, usando a base 256 (16^2), pois ele não cabe em apenas uma posição. Se ele for menor que 65536 (256^2) devemos "quebrá-lo" em dois, se for maior que 65535 e menor que 16777216 (256^3), devemos quebrá-lo em três e assim por diante.

Suponha, por exemplo, que você queira colocar o número 522 nas memórias 3ØØØØ e 3ØØØ1; ora, 522 = 'Ø2 ¦ ØA' { 'Ø2' = 2; 'ØA' = 1Ø }

Vamos colocar o byte menos significativo na memória de menor endereço:

```
POKE 3ØØØØ,1Ø
POKE 3ØØØ1,2
```

Para obtê-lo de volta, basta fazer:

```
PRINT PEEK 3ØØØØ + 256 * PEEK 3ØØØ1
```

Experimente agora colocar na memória o número 32477 e obtê-lo de volta (use os mesmos endereços).

Faça o mesmo para o número 65535... e, se você estiver inspirado, tente colocar os números 1ØØØØØ e 2ØØØØØØØ na memória.

TABELAS DAS INSTRUÇÕES DO ASSEMBLY Z8Ø E CARACTERES DO DO TK 82(85)

TABELA DE CARACTERES DO TK

CÓDIGO	HEXADECIMAL	CARACTERE
ØØØ	'ØØ'	
ØØ1	'Ø1'	
ØØ2	'Ø2'	
ØØ3	'Ø3'	
ØØ4	'Ø4'	
ØØ5	'Ø5'	
ØØ6	'Ø6'	
ØØ7	'Ø7'	
ØØ8	'Ø8'	
ØØ9	'Ø9'	
Ø1Ø	'ØA'	
Ø11	'ØB'	"
Ø12	'ØC'	£
Ø13	'ØD'	$
Ø14	'ØE'	:
Ø15	'ØF'	?
Ø16	'1Ø'	(
Ø17	'11'	)
Ø18	'12'	>
Ø19	'13'	<
Ø2Ø	'14'	=

CÓDIGO	HEXADECIMAL	CARACTERE
Ø21	'15'	+
Ø22	'16'	–
Ø23	'17'	*
Ø24	'18'	/
Ø25	'19'	;
Ø26	'1A'	,
Ø27	'1B'	.
Ø28	'1C'	Ø
Ø29	'1D'	1
Ø3Ø	'1E'	2
Ø31	'1F'	3
Ø32	'2Ø'	4
Ø33	'21'	5
Ø34	'22'	6
Ø35	'23'	7
Ø36	'24'	8
Ø37	'25'	9
Ø38	'26'	A
Ø39	'27'	B
Ø4Ø	'28'	C
Ø41	'29'	D
Ø42	'2A'	E
Ø43	'2B'	F
Ø44	'2C'	G
Ø45	'2D'	H
Ø46	'2E'	I
Ø47	'2F'	J
Ø48	'3Ø'	K
Ø49	'31'	L
Ø5Ø	'32'	M

CÓDIGO	HEXADECIMAL	CARACTERE
Ø51	'33'	N
Ø52	'34'	O
Ø53	'35'	P
Ø54	'36'	Q
Ø55	'37'	R
Ø56	'38'	S
Ø57	'39'	T
Ø58	'3A'	U
Ø59	'3B'	V
Ø6Ø	'3C'	W
Ø61	'3D'	X
Ø62	'3E'	Y
Ø63	'3F'	Z
Ø64	'4Ø'	RND
Ø65	'41'	INKEY$
Ø66	'42'	PI
Ø67	'43'	?
Ø68	'44'	?
Ø69	'45'	?
Ø7Ø	'46'	?
Ø71	'47'	?
Ø72	'48'	?
Ø73	'49'	?
Ø74	'4A'	?
Ø75	'4B'	?
Ø76	'4C'	?
Ø77	'4D'	?
Ø78	'4E'	?
Ø79	'4F'	?
Ø8Ø	'5Ø'	?

CÓDIGO	HEXADECIMAL	CARACTERE
Ø81	'51'	?
Ø82	'52'	?
Ø83	'53'	?
Ø84	'54'	?
Ø85	'55'	?
Ø86	'56'	?
Ø87	'57'	?
Ø88	'58'	?
Ø89	'59'	?
Ø9Ø	'5A'	?
Ø91	'5B'	?
Ø92	'5C'	?
Ø93	'5D'	?
Ø94	'5E'	?
Ø95	'5F'	?
Ø96	'6Ø'	?
Ø97	'61'	?
Ø98	'62'	?
Ø99	'63'	?
1ØØ	'64'	?
1Ø1	'65'	?
1Ø2	'66'	?
1Ø3	'67'	?
1Ø4	'68'	?
1Ø5	'69'	?
1Ø6	'6A'	?
1Ø7	'6B'	?
1Ø8	'6C'	?
1Ø9	'6D'	?
11Ø	'6E'	?
111	'6F'	?

CÓDIGO	HEXADECIMAL	CARACTERE
112	'7Ø'	?
113	'71'	?
114	'72'	?
115	'73'	?
116	'74'	?
117	'75'	?
118	'76'	?
119	'77'	?
12Ø	'78'	?
121	'79'	?
122	'7A'	?
123	'7B'	?
124	'7C'	?
125	'7D'	?
126	'7E'	?
127	'7F'	?
128	'8Ø'	
129	'81'	
13Ø	'82'	
131	'83'	
132	'84'	
133	'85'	
134	'86'	
135	'87'	
136	'88'	
137	'89'	
138	'8A'	
139	'8B'	''
14Ø	'8C'	£
141	'8D'	$

CÓDIGO	HEXADECIMAL	CARACTERE
142	'8E'	:
143	'8F'	?
144	'9Ø'	(
145	'91'	)
146	'92'	>
147	'93'	<
148	'94'	=
149	'95'	+
15Ø	'96'	–
151	'97'	*
152	'98'	/
153	'99'	;
154	'9A'	,
155	'9B'	.
156	'9C'	Ø
157	'9D'	1
158	'9E'	2
159	'9F'	3
16Ø	'AØ'	4
161	'A1'	5
162	'A2'	6
163	'A3'	7
164	'A4'	8
165	'A5'	9
166	'A6'	A
167	'A7'	B
168	'A8'	C
169	'A9'	D
17Ø	'AA'	E
171	'AB'	F

CÓDIGO	HEXADECIMAL	CARACTERE
172	'AC'	G
173	'AD'	H
174	'AE'	I
175	'AF'	J
176	'BØ'	K
177	'B1'	L
178	'B2'	M
179	'B3'	N
18Ø	'B4'	O
181	'B5'	P
182	'B6'	Q
183	'B7'	R
184	'B8'	S
185	'B9'	T
186	'BA'	U
187	'BB'	V
188	'BC'	W
189	'BD'	X
19Ø	'BE'	Y
191	'BF'	Z
192	'CØ'	" "
193	'C1'	AT
194	'C2'	TAB
195	'C3'	?
196	'C4'	CODE
197	'C5'	VAL
198	'C6'	LEN
199	'C7'	SIN
2ØØ	'C8'	COS
2Ø1	'C9'	TAN

CÓDIGO	HEXADECIMAL	CARACTERE
2Ø2	'CA'	ASN
2Ø3	'CB'	ACS
2Ø4	'CC'	ATN
2Ø5	'CD'	LN
2Ø6	'CE'	EXP
2Ø7	'CF'	INT
2Ø8	'DØ'	SQR
2Ø9	'D1'	SGN
21Ø	'D2'	ABS
211	'D3'	PEEK
212	'D4'	USR
213	'D5'	STR$
214	'D6'	CHR$
215	'D7'	NOT
216	'D8'	**
217	'D9'	OR
218	'DA'	AND
219	'DB'	<=
22Ø	'DC'	>=
221	'DD'	<>
222	'DE'	THEN
223	'DF'	TO
224	'EØ'	STEP
225	'E1'	LPRINT
226	'E2'	LLIST
227	'E3'	STOP
228	'E4'	SLOW
229	'E5'	FAST
23Ø	'E6'	NEW
231	'E7'	SCROLL

CÓDIGO	HEXADECIMAL	CARACTERE
232	'E8'	CONT
233	'E9'	DIM
234	'EA'	REM
235	'EB'	FOR
236	'EC'	GOTO
237	'ED'	GOSUB
238	'EE'	INPUT
239	'EF'	LOAD
240	'F0'	LIST
241	'F1'	LET
242	'F2'	PAUSE
243	'F3'	NEXT
244	'F4'	POKE
245	'F5'	PRINT
246	'F6'	PLOT
247	'F7'	RUN
248	'F8'	SAVE
249	'F9'	RAND
250	'FA'	IF
251	'FB'	CLS
252	'FC'	UNPLOT
253	'FD'	CLEAR
254	'FE'	RETURN
255	'FF'	COPY

Observação: Os caracteres envoltos por um quadrado significam caractere em vídeo-reverso.

TABELA DAS INSTRUÇÕES EM LINGUAGEM DE MÁQUINA DIVIDIDAS POR GRUPOS

– MNEMÔNICOS
– NÚMERO DE BYTES
– FLAGS AFETADAS
– CÓDIGO HEXADECIMAL
– CARACTERE PRINCIPAL CORRESPONDENTE

Este apêndice apresenta uma tabela muito útil de todas as instruções da linguagem de máquina disponíveis para o microprocessador do TK (modelo Z80), divididas em grupos funcionais, a saber:

- instruções de deslocamento;
- instruções aritméticas – incremento e decremento;
- o STACK POINTER;
- instruções de salto e sub-rotinas – instruções de comparação;
- instruções lógicas e operações com bits – instruções de "rotação";
- instruções de interrupção – instruções de entrada e saída;
- outras instruções.

Para cada instrução é apresentado o número de bytes, seu código hexadecimal e as flags afetadas com a seguinte notação:

*	:	flag afetada; resultado depende da operação
?	:	flag afetada; resultado indefinido
0	:	flag afetada; resultado zero
1	:	flag afetada; resultado um
–	:	flag não afetada; permanece em seu valor anterior à instrução
*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, são apresentados o código decimal e o caractere correspondentes ao byte principal da instrução, isto é, o byte que *não seja dado, endereço ou prefixo.* Desse modo, é conveniente lembrar:

Código	*Decimal*	*Caractere*		
'CB'	203	ACS	:	prefixo instruções SET/RESET/BIT/rotação
'ED'	237	GOSUB	:	prefixo de várias instruções
'DD'	221	<>	:	prefixo de instruções usando IX
'FD'	253	CLEAR	:	prefixo de instruções usando IY

Notações genéricas utilizadas:

r. . . . . .registro (A, B, C, D, H ou L)
rr . . . . .par de registros (BC, DE, HL)

HXY. . . . . .pares HL, IX ou IY
K . . . . . . .número hexadecimal de 1 byte (2 algarismos)
KK. . . . . .número hexadecimal de 2 bytes (4 algarismos)
Q . . . . . . .número hexadecimal de 1 byte com sinal (2 algarismos: –127 a 128 em decimal)
(cond). . . .condições: C, NC, Z, NZ, PE, PO, M ou P
(repetição) . . . . D, DR, I ou IR
b . . . . número de Ø a 7 (bits)

A) INSTRUÇÕES DE DESLOCAMENTO

Notação genérica:

LD r,r
LD A,I
LD I,A
LD A,R
LD R,A
LD SP,HXY
LD r,K
LD rr,KK
LD HXY,KK
LD SP,KK
LD (HXY + Q),K (+Q não vale para HL)
LD A,(BC)
LD A,(DE)
LD r,(HXY + Q) (+Q não vale para HL)
LD (BC),A
LD (DE),A
LD (HXY + Q),r (+Q não vale para HL)
LD A,(KK)
LD rr,(KK)
LD HXY,(KK)
LD (KK),A
LD (KK),rr
LD (KK),HXY
LD SP,(KK)
LD (KK),SP
LD (repetição)

Observação: Neste último caso, para LDI e LDD, P/O = 1 se BC ≠ Ø; senão, P/O = Ø.

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
LD A,A	1	–	–	–	–	–	–	'7F'	127	cursor
LD A,B	1	–	–	–	–	–	–	'78'	12Ø	modo
LD A,C	1	–	–	–	–	–	–	'79'	121	function
LD A,D	1	–	–	–	–	–	–	'7A'	122	
LD A,E	1	–	–	–	–	–	–	'7B'	123	
LD A,H	1	–	–	–	–	–	–	'7C'	124	
LD A,L	1	–	–	–	–	–	–	'7D'	125	
LD B,A	1	–	–	–	–	–	–	'47'	71	
LD B,B	1	–	–	–	–	–	–	'4Ø'	64	RND
LD B,C	1	–	–	–	–	–	–	'41'	65	INKEY$
LD B,D	1	–	–	–	–	–	–	'42'	66	PI
LD B,E	1	–	–	–	–	–	–	'43'	67	
LD B,H	1	–	–	–	–	–	–	'44'	68	
LD B,L	1	–	–	–	–	–	–	'45'	69	
LD C,A	1	–	–	–	–	–	–	'4F'	79	
LD C,B	1	–	–	–	–	–	–	'48'	72	
LD C,C	1	–	–	–	–	–	–	'49'	73	
LD C,D	1	–	–	–	–	–	–	'4A'	74	
LD C,E	1	–	–	–	–	–	–	'4B'	75	
LD C,H	1	–	–	–	–	–	–	'4C'	76	
LD C,L	1	–	–	–	–	–	–	'4D'	77	
LD D,A	1	–	–	–	–	–	–	'57'	87	
LD D,B	1	–	–	–	–	–	–	'5Ø'	8Ø	
LD D,C	1	–	–	–	–	–	–	'51'	81	
LD D,D	1	–	–	–	–	–	–	'52'	82	
LD D,E	1	–	–	–	–	–	–	'53'	83	
LD D,H	1	–	–	–	–	–	–	'54'	84	
LD D,L	1	–	–	–	–	–	–	'55'	85	
LD E,A	1	–	–	–	–	–	–	'5F'	95	

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
LD E,B	1	—	—	—	—	—	—	'58'	88	
LD E,C	1	—	—	—	—	—	—	'59'	89	
LD E,D	1	—	—	—	—	—	—	'5A'	9Ø	
LD E,E	1	—	—	—	—	—	—	'5B'	91	
LD E,H	1	—	—	—	—	—	—	'5C'	92	
LD E,L	1	—	—	—	—	—	—	'5D'	93	
LD H,A	1	—	—	—	—	—	—	'67'	1Ø3	
LD H,B	1	—	—	—	—	—	—	'6Ø'	96	
LD H,C	1	—	—	—	—	—	—	'61'	97	
LD H,D	1	—	—	—	—	—	—	'62'	98	
LD H,E	1	—	—	—	—	—	—	'63'	99	
LD H,H	1	—	—	—	—	—	—	'64'	1ØØ	
LD H,L	1	—	—	—	—	—	—	'65'	1Ø1	
LD L,A	1	—	—	—	—	—	—	'6F'	111	
LD L,B	1	—	—	—	—	—	—	'68'	1Ø4	
LD L,C	1	—	—	—	—	—	—	'69'	1Ø5	
LD L,D	1	—	—	—	—	—	—	'6A'	1Ø6	
LD L,E	1	—	—	—	—	—	—	'6B'	1Ø7	
LD L,H	1	—	—	—	—	—	—	'6C'	1Ø8	
LD L,L	1	—	—	—	—	—	—	'6D'	1Ø9	
LD A,I	2	—	*	*	T	Ø	Ø	'ED57'	87	
LD I,A	2	—	—	—	—	—	—	'ED47'	71	
LD A,R	2	—	*	*	T	Ø	Ø	'ED5F'	95	
LD R,A	2	—	—	—	—	—	—	'ED4F'	79	
LD SP,HL	1	—	—	—	—	—	—	'F9'	249	RAND
LD SP,IX	2	—	—	—	—	—	—	'DDF9'	249	RAND
LD SP,IY	2	—	—	—	—	—	—	'FDF9'	249	RAND
LD A,K	2	—	—	—	—	—	—	'3EK'	62	Y
LD B,K	2	—	—	—	—	—	—	'Ø6K'	6	

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
LD C,K	2	—	—	—	—	—	—	'ØEK'	14	:
LD D,K	2	—	—	—	—	—	—	'16K'	22	—
LD E,K	2	—	—	—	—	—	—	'1EK'	3Ø	2
LD H,K	2	—	—	—	—	—	—	'26K'	38	A
LD L,K	2	—	—	—	—	—	—	'2EK'	46	I
LD BC,KK	3	—	—	—	—	—	—	'Ø1KK'	1	
LD DE,KK	3	—	—	—	—	—	—	'11KK'	17	)
LD HL,KK	3	—	—	—	—	—	—	'21KK'	33	5
LD IX,KK	4	—	—	—	—	—	—	'DD21KK'	33	5
LD IY,KK	4	—	—	—	—	—	—	'FD21KK'	33	5
LD SP,KK	3	—	—	—	—	—	—	'31KK'	49	L
LD (HL),K	2	—	—	—	—	—	—	'36K'	54	Q
LD (IX+Q),K	4	—	—	—	—	—	—	'DD36QK'	54	Q
LD (IY+Q),K	4	—	—	—	—	—	—	'FD36QK'	54	Q
LD A,(BC)	1	—	—	—	—	—	—	'ØA'	1Ø	
LD A,(DE)	1	—	—	—	—	—	—	'1A'	26	
LD A,(HL)	1	—	—	—	—	—	—	'7E'	126	número
LD A,(IX+Q)	3	—	—	—	—	—	—	'DD7EQ'	126	número
LD A,(IY+Q)	3	—	—	—	—	—	—	'FD7EQ'	126	número
LD B,(HL)	1	—	—	—	—	—	—	'46'	7Ø	
LD B,(IX+Q)	3	—	—	—	—	—	—	'DD46Q'	7Ø	
LD B,(IY+Q)	3	—	—	—	—	—	—	'FD46Q'	7Ø	
LD C,(HL)	1	—	—	—	—	—	—	'4E'	78	
LD C,(IX+Q)	3	—	—	—	—	—	—	'DD4EQ'	78	
LD C,(IY+Q)	3	—	—	—	—	—	—	'FD4EQ'	78	
LD D,(HL)	1	—	—	—	—	—	—	'56'	86	
LD D,(IX+Q)	3	—	—	—	—	—	—	'DD56Q'	86	
LD D,(IY+Q)	3	—	—	—	—	—	—	'FD56Q'	86	

Instrução	Número de bytes	C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
LD E,(HL)	1	–	–	–	–	–	–	'5E'	94	
LD E,(IX+Q)	3	–	–	–	–	–	–	'DD5EQ'	94	
LD E,(IY+Q)	3	–	–	–	–	–	–	'FD5EQ'	94	
LD H,(HL)	1	–	–	–	–	–	–	'66'	1Ø2	
LD H,(IX+Q)	3	–	–	–	–	–	–	'DD66Q'	1Ø2	
LD H,(IY+Q)	3	–	–	–	–	–	–	'FD66Q'	1Ø2	
LD L,(HL)	1	–	–	–	–	–	–	'6E'	11Ø	
LD L,(IX+Q)	3	–	–	–	–	–	–	'DD6EQ'	11Ø	
LD L,(IY+Q)	3	–	–	–	–	–	–	'FD6EQ'	11Ø	
LD (BC),A	1	–	–	–	–	–	–	'Ø2'	2	▝
LD (DE),A	1	–	–	–	–	–	–	'12'	18	>
LD (HL),A	1	–	–	–	–	–	–	'77'	119	RUBOUT
LD (IX+Q),A	3	–	–	–	–	–	–	'DD77Q'	119	RUBOUT
LD (IY+Q),A	3	–	–	–	–	–	–	'FD77Q'	119	RUBOUT
LD (HL),B	1	–	–	–	–	–	–	'7Ø'	112	⇧
LD (IX+Q),B	3	–	–	–	–	–	–	'DD7ØQ'	112	⇧
LD (IY+Q),B	3	–	–	–	–	–	–	'FD7ØQ'	112	⇧
LD (HL),C	1	–	–	–	–	–	–	'71'	113	⇩
LD (IX+Q),C	3	–	–	–	–	–	–	'DD71Q'	113	⇩
LD (IY+Q),C	3	–	–	–	–	–	–	'FD71Q'	113	⇩
LD (HL),D	1	–	–	–	–	–	–	'72'	114	⇦
LD (IX+Q),D	3	–	–	–	–	–	–	'DD72Q'	114	⇦
LD (IY+Q),D	3	–	–	–	–	–	–	'FD72Q'	114	⇦
LD (HL),E	1	–	–	–	–	–	–	'73'	115	⇨
LD (IX+Q),E	3	–	–	–	–	–	–	'DD73Q'	115	⇨
LD (IY+Q),E	3	–	–	–	–	–	–	'FD73Q'	115	⇨

Instrução	Número de bytes	C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
LD (HL),H	1	–	–	–	–	–	–	'74'	116	GRAPHICS
LD (IX+Q),H	3	–	–	–	–	–	–	'DD74Q'	116	GRAPHICS
LD (IY+Q),H	3	–	–	–	–	–	–	'FD74Q'	116	GRAPHICS
LD (HL),L	1	–	–	–	–	–	–	'75'	117	EDIT
LD (IX+Q),L	3	–	–	–	–	–	–	'DD75Q'	117	EDIT
LD (IY+Q),L	3	–	–	–	–	–	–	'FD75Q'	117	EDIT
LD A,(KK)	3	–	–	–	–	–	–	'3AKK'	58	U
LD BC,(KK)	4	–	–	–	–	–	–	'ED4BKK'	75	
LD DE,(KK)	4	–	–	–	–	–	–	'ED5BKK'	91	
LD HL,(KK)	3	–	–	–	–	–	–	'2AKK'	42	E
LD IX,(KK)	4	–	–	–	–	–	–	'DD2AKK'	42	E
LD IY,(KK)	4	–	–	–	–	–	–	'FD2AKK'	42	E
LD (KK),A	3	–	–	–	–	–	–	'32KK'	5Ø	M
LD (KK),BC	4	–	–	–	–	–	–	'ED43KK'	67	
LD (KK),DE	4	–	–	–	–	–	–	'ED53KK'	83	
LD (KK),HL	3	–	–	–	–	–	–	'22KK'	34	6
LD (KK),IX	4	–	–	–	–	–	–	'DD22KK'	34	6
LD (KK),IY	4	–	–	–	–	–	–	'FD22KK'	34	6
LD SP,(KK)	4	–	–	–	–	–	–	'ED7BKK'	123	
LD (KK),SP	4	–	–	–	–	–	–	'ED73KK'	115	⇨
LDD	2	–	–	–	*	Ø	Ø	'EDA8'	168	C
LDDR	2	–	–	–	Ø	Ø	Ø	'EDB8'	184	S
LDI	2	–	–	–	*	Ø	Ø	'EDAØ'	16Ø	4
LDIR	2	–	–	–	Ø	Ø	Ø	'EDBØ'	176	K

Alexandre Fernandes
RG: 14.517.733

B) **INSTRUÇÕES ARITMÉTICAS; INCREMENTO E DECREMENTO**

Notação genérica:

```
ADD   A,r
ADD   A,(HXY+Q)
ADC   A,r
ADC   A,(HXY+Q)
ADD   HL,rr
ADD   IX, ⎧BC
          ⎨DE
          ⎩IX
ADD   IY, ⎧BC
          ⎨DE
          ⎩IY
ADD   HXY,SP
ADC   HL,rr
ADC   HL,SP
ADD   A,K
ADC   A,K
SUB   A,r
SUB   A,(HXY+Q)
SBC   A,r
SBC   A,(HXY+Q)
SBC   HL,rr
SBC   HL,SP
SUB   A,K
SBC   A,K
INC   r
INC   rr
INC   HXY
INC   SP
INC   (HXY+Q)
DEC   r
DEC   rr
DEC   HXY
DEC   SP
DEC   (HXY+Q)
NEG
```

Observação: Nas instruções com (HXY + Q), +Q não vale para HL.

Instrução	Número de bytes	Flags						Código Hexadecimal	Código Decimal	Caracteres
		C	Z	S	P/O	AC	N			
ADD A,A	1	*	*	*	*o	*	Ø	'87'	135	
ADD A,B	1	*	*	*	*o	*	Ø	'8Ø'	128	
ADD A,C	1	*	*	*	*o	*	Ø	'81'	129	
ADD A,D	1	*	*	*	*o	*	Ø	'82'	13Ø	
ADD A,E	1	*	*	*	*o	*	Ø	'83'	131	
ADD A,H	1	*	*	*	*o	*	Ø	'84'	132	
ADD A,L	1	*	*	*	*o	*	Ø	'85'	133	
ADD A,(HL)	1	*	*	*	*o	*	Ø	'86'	134	
ADD A,(IX+Q)	3	*	*	*	*o	*	Ø	'DD86Q'	134	
ADD A,(IY+Q)	3	*	*	*	*o	*	Ø	'FD86Q'	134	
ADC A,A	1	*	*	*	*o	*	Ø	'8F'	143	?
ADC A,B	1	*	*	*	*o	*	Ø	'88'	136	
ADC A,C	1	*	*	*	*o	*	Ø	'89'	137	
ADC A,D	1	*	*	*	*o	*	Ø	'8A'	138	
ADC A,E	1	*	*	*	*o	*	Ø	'8B'	139	"
ADC A,H	1	*	*	*	*o	*	Ø	'8C'	14Ø	£
ADC A,L	1	*	*	*	*o	*	Ø	'8D'	141	$
ADC A,(HL)	1	*	*	*	*o	*	Ø	'8E'	142	:
ADC A,(IX+Q)	3	*	*	*	*o	*	Ø	'DD8EQ'	142	:
ADC A,(IY+Q)	3	*	*	*	*o	*	Ø	'FD8EQ'	142	:
ADD HL,BC	1	*	—	—	—	?	Ø	'Ø9'	9	
ADD HL,DE	1	*	—	—	—	?	Ø	'19'	25	;
ADD HL,HL	1	*	—	—	—	?	Ø	'29'	41	D
ADD IX,BC	2	*	—	—	—	?	Ø	'DDØ9'	9	
ADD IX,DE	2	*	—	—	—	?	Ø	'DD19'	25	;
ADD IX,IX	2	*	—	—	—	?	Ø	'DD29'	41	D
ADD IY,BC	2	*	—	—	—	?	Ø	'FDØ9'	9	
ADD IY,DE	2	*	—	—	—	?	Ø	'FD19'	25	;
ADD IY,IY	2	*	—	—	—	?	Ø	'FD29'	41	D
ADD HL,SP	1	*	—	—	—	?	Ø	'39'	57	T
ADD IX,SP	2	*	—	—	—	?	Ø	'DD39'	57	T

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
ADD IY,SP	2	*	–	–	–	?	Ø	'FD39'	57	T
ADC HL,BC	2	*	*	*	*o	?	Ø	'ED4A'	74	
ADC HL,DE	2	*	*	*	*o	?	Ø	'ED5A'	9Ø	
ADC HL,HL	2	*	*	*	*o	?	Ø	'ED6A'	1Ø6	
ADC HL,SP	2	*	*	*	*o	?	Ø	'ED7A'	122	
ADD A,K	2	*	*	*	*o	*	Ø	'C6K'	198	LEN
ADC A,K	2	*	*	*	*o	*	Ø	'CEK'	2Ø6	EXP
SUB A,A	1	*	*	*	*o	*	1	'97'	151	*
SUB A,B	1	*	*	*	*o	*	1	'9Ø'	144	(
SUB A,C	1	*	*	*	*o	*	1	'91'	145	)
SUB A,D	1	*	*	*	*o	*	1	'92'	146	>
SUB A,E	1	*	*	*	*o	*	1	'93'	147	<
SUB A,H	1	*	*	*	*o	*	1	'94'	148	=
SUB A,L	1	*	*	*	*o	*	1	'95'	149	+
SUB A,(HL)	1	*	*	*	*o	*	1	'96'	15Ø	–
SUB A,(IX+Q)	3	*	*	*	*o	*	1	'DD96Q'	15Ø	–
SUB A,(IY+Q)	3	*	*	*	*o	*	1	'FD96Q'	15Ø	–
SBC A,A	1	*	*	*	*o	*	1	'9F'	159	3
SBC A,B	1	*	*	*	*o	*	1	'98'	152	/
SBC A,C	1	*	*	*	*o	*	1	'99'	153	;
SBC A,D	1	*	*	*	*o	*	1	'9A'	154	,
SBC A,E	1	*	*	*	*o	*	1	'9B'	155	.
SBC A,H	1	*	*	*	*o	*	1	'9C'	156	Ø
SBC A,L	1	*	*	*	*o	*	1	'9D'	157	1
SBC A,(HL)	1	*	*	*	*o	*	1	'9E'	158	2
SBC A,(IX+Q)	3	*	*	*	*o	*	1	'DD9EQ'	158	2
SBC A,(IY+Q)	3	*	*	*	*o	*	1	'FD9EQ'	158	2
SBC HL,BC	2	*	*	*	*o	?	1	'ED42'	66	PI
SBC HL,DE	2	*	*	*	*o	?	1	'ED52'	82	

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
SBC HL,HL	2	*	*	*	*o	?	1	'ED62'	98	
SBC HL,SP	2	*	*	*	*o	?	1	'ED72'	114	⇦
SUB A,K	2	*	*	*	*o	*	1	'D6K'	214	CHR$
SBC A,K	2	*	*	*	*o	*	1	'DEK'	222	THEN
INC A	1	–	*	*	*o	*	Ø	'3C'	6Ø	W
INC B	1	–	*	*	*o	*	Ø	'Ø4'	4	▖
INC C	1	–	*	*	*o	*	Ø	'ØC'	12	£
INC D	1	–	*	*	*o	*	Ø	'14'	2Ø	=
INC E	1	–	*	*	*o	*	Ø	'1C'	28	Ø
INC H	1	–	*	*	*o	*	Ø	'24'	36	8
INC L	1	–	*	*	*o	*	Ø	'2C'	44	G
INC BC	1	–	–	–	–	–	–	'Ø3'	3	▀
INC DE	1	–	–	–	–	–	–	'13'	19	<
INC HL	1	–	–	–	–	–	–	'23'	35	7
INC IX	2	–	–	–	–	–	–	'DD23'	35	7
INC IY	2	–	–	–	–	–	–	'FD23'	35	7
INC SP	1	–	–	–	–	–	–	'33'	51	N
INC (HL)	1	–	*	*	*o	*	Ø	'34'	52	O
INC (IX+Q)	3	–	*	*	*o	*	Ø	'DD34Q'	52	O
INC (IY+Q)	3	–	*	*	*o	*	Ø	'FD34Q'	52	O
DEC A	1	–	*	*	*o	*	1	'3D'	61	X
DEC B	1	–	*	*	*o	*	1	'Ø5'	5	▌
DEC C	1	–	*	*	*o	*	1	'ØD'	13	$
DEC D	1	–	*	*	*o	*	1	'15'	21	+
DEC E	1	–	*	*	*o	*	1	'1D'	29	1
DEC H	1	–	*	*	*o	*	1	'25'	37	9
DEC L	1	–	*	*	*o	*	1	'2D'	45	H
DEC BC	1	–	–	–	–	–	–	'ØB'	11	"
DEC DE	1	–	–	–	–	–	–	'1B'	27	.

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
DEC HL	1	–	–	–	–	–	–	'2B'	43	F
DEC IX	2	–	–	–	–	–	–	'DD2B'	43	F
DEC IY	2	–	–	–	–	–	–	'FD2B'	43	F
DEC SP	1	–	–	–	–	–	–	'3B'	59	V
DEC (HL)	1	–	*	*	*o	*	1	'35'	53	P
DEC (IX+Q)	3	–	*	*	*o	*	1	'DD35Q'	53	P
DEC (IY+Q)	3	–	*	*	*o	*	1	'FD35Q'	53	P
NEG	2	*	*	*	*o	*	1	'ED44'	68	

C) **O STACK POINTER**

Notação genérica:

```
PUSH  AF
PUSH  rr
PUSH  (HXY)
POP   AF
POP   rr
POP   (HXY)
LD    SP,HXY
LD    SP,KK
LD    SP,(KK)
LD    (KK),SP
ADD   HXY,SP
ADC   HL,SP
SBC   HL,SP
INC   SP
DEC   SP
EX    (SP),HXY
```


Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
PUSH AF	1	–	–	–	–	–	–	'F5'	245	PRINT
PUSH BC	1	–	–	–	–	–	–	'C5'	197	VAL
PUSH DE	1	–	–	–	–	–	–	'D5'	213	STR$
PUSH HL	1	–	–	–	–	–	–	E5'	229	FAST
PUSH IX	2	–	–	–	–	–	–	'DDE5'	229	FAST
PUSH IY	2	–	–	–	–	–	–	'FDE5'	229	FAST
POP AF	1	–	–	–	–	–	–	'F1'	241	LET
POP BC	1	–	–	–	–	–	–	'C1	193	AT
POP DE	1	–	–	–	–	–	–	'D1'	209	SGN
POP HL	1	–	–	–	–	–	–	'E1'	225	LPRINT
POP IX	2	–	–	–	–	–	–	'DDE1'	225	LPRINT
POP IY	2	–	–	–	–	–	–	'FDE1'	225	LPRINT
LD SP,HL	1	–	–	–	–	–	–	'F9'	249	RAND
LD SP,IX	2	–	–	–	–	–	–	'DDF9'	249	RAND
LD SP,IY	2	–	–	–	–	–	–	'FDF9'	249	RAND
LD SP,KK	3	–	–	–	–	–	–	'31KK'	49	L
LD SP,(KK)	4	–	–	–	–	–	–	'ED7BKK'	123	
LD (KK),SP	4	–	–	–	–	–	–	'ED73KK'	115	⇨
ADD HL,SP	1	*	–	–	–	?	Ø	'39'	57	T
ADD IX,SP	2	*	–	–	–	?	Ø	'DD39'	57	T
ADD IY,SP	2	*	–	–	–	?	Ø	'FD39'	57	T
ADC HL,SP	2	*	*	*	*o	?	Ø	'ED7A'	122	
SBC HL,SP	2	*	*	*	*o	?	1	'ED72'	114	⇦
INC SP	1	–	–	–	–	–	–	'33'	51	N
DEC SP	1	–	–	–	–	–	–	'3B'	59	V
EX (SP),HL	1	–	–	–	–	–	–	'E3'	227	STOP
EX (SP),IX	2	–	–	–	–	–	–	'DDE3'	227	STOP
EX (SP),IY	2	–	–	–	–	–	–	'FDE3'	227	STOP

D) INSTRUÇÕES DE SALTO E SUB-ROTINAS; INSTRUÇÕES DE COMPARAÇÃO

Notação genérica:

JP KK
JP (HXY)
JP {cond.} , KK
JR Q
JR {C, NC, Z, NZ}, Q
DJNZ Q
CALL KK
CALL {cond.}, KK
RET
RET {cond.}
RETI
RETN
RST {Ø, 8, 16, 24, 32, 4Ø, 48, 56}

CP r
CP (HXY + Q) (+Q não vale para HL)
CP K
CP (repetição)

Observação: Neste último caso, Z = 1 se A = (HL); P/O = 1 se BC ≠ Ø senão P/O = Ø.

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
JP KK	3	–	–	–	–	–	–	'C3KK'	195	
JP (HL)	1	–	–	–	–	–	–	'E9'	233	DIM
JP (IX)	2	–	–	–	–	–	–	'DDE9'	233	DIM
JP (IY)	2	–	–	–	–	–	–	'FDE9'	233	DIM
JP C,KK	3	–	–	–	–	–	–	'DAKK'	218	AND
JP NC,KK	3	–	–	–	–	–	–	'D2KK'	21Ø	ABS
JP Z,KK	3	–	–	–	–	–	–	'CAKK'	2Ø2	ASN
JP NZ,KK	3	–	–	–	–	–	–	'C2KK'	194	TAB
JP PE,KK	3	–	–	–	–	–	–	'EAKK'	234	REM
JP PO,KK	3	–	–	–	–	–	–	'E2KK'	226	LLIST
JP M,KK	3	–	–	–	–	–	–	'FAKK'	25Ø	IF
JP P,KK	3	–	–	–	–	–	–	'F2KK'	242	PAUSE
JR Q	2	–	–	–	–	–	–	'18Q'	24	/
JR C,Q	2	–	–	–	–	–	–	'38Q'	56	S
JR NC,Q	2	–	–	–	–	–	–	'3ØQ'	48	K
JR Z,Q	2	–	–	–	–	–	–	'28Q'	4Ø	C
JR NZ,Q	2	–	–	–	–	–	–	'2ØQ'	32	4
DJNZ Q	2	–	–	–	–	–	–	'1ØQ'	16	(
CALL KK	3	–	–	–	–	–	–	'CDKK'	2Ø5	LN
CALL C,KK	3	–	–	–	–	–	–	'DCKK'	22Ø	>=
CALL NC,KK	3	–	–	–	–	–	–	'D4KK'	212	USR
CALL Z,KK	3	–	–	–	–	–	–	'CCKK'	2Ø4	ATN
CALL NZ,KK	3	–	–	–	–	–	–	'C4KK'	196	CODE
CALL PE,KK	3	–	–	–	–	–	–	'ECKK'	236	GOTO
CALL PO,KK	3	–	–	–	–	–	–	'E4KK'	228	SLOW
CALL M,KK	3	–	–	–	–	–	–	'FCKK'	252	UNPLOT
CALL P,KK	3	–	–	–	–	–	–	'F4KK'	244	POKE
RET	1	–	–	–	–	–	–	'C9'	2Ø1	TAN
RET C	1	–	–	–	–	–	–	'D8'	216	**

Instrução	Número de bytes	Flags						Código Hexadecimal	Código Decimal	Caracteres
		C	Z	S	P/O	AC	N			
RET NC	1	–	–	–	–	–	–	'DØ'	2Ø8	SQR
RET Z	1	–	–	–	–	–	–	'C8'	2ØØ	COS
RET NZ	1	–	–	–	–	–	–	'CØ'	192	" "
RET PE	1	–	–	–	–	–	–	'E8'	232	CONT
RET PO	1	–	–	–	–	–	–	'EØ'	224	STEP
RET M	1	–	–	–	–	–	–	'F8'	248	SAVE
RET P	1	–	–	–	–	–	–	'FØ'	24Ø	LIST
RETI	2	–	–	–	–	–	–	'ED4D'	77	
RETN	2	–	–	–	–	–	–	'ED45'	69	
RST Ø	1	–	–	–	–	–	–	'C7'	199	SIN
RST 8	1	–	–	–	–	–	–	'CF'	2Ø7	INT
RST 16	1	–	–	–	–	–	–	'D7'	215	NOT
RST 24	1	–	–	–	–	–	–	'DF'	223	TO
RST 32	1	–	–	–	–	–	–	'E7'	231	SCROLL
RST 4Ø	1	–	–	–	–	–	–	'EF'	239	LOAD
RST 48	1	–	–	–	–	–	–	'F7'	247	RUN
RST 56	1	–	–	–	–	–	–	'FF'	255	COPY
CP A	1	*	*	*	*o	*	1	'BF'	191	Z
CP B	1	*	*	*	*o	*	1	'B8'	184	S
CP C	1	*	*	*	*o	*	1	'B9'	185	T
CP D	1	*	*	*	*o	*	1	'BA'	186	U
CP E	1	*	*	*	*o	*	1	'BB'	187	V
CP H	1	*	*	*	*o	*	1	'BC'	188	W
CP L	1	*	*	*	*o	*	1	'BD'	189	X
CP (HL)	1	*	*	*	*o	*	1	'BE'	19Ø	Y
CP (IX+Q)	3	*	*	*	*o	*	1	'DDBEQ'	19Ø	Y
CP (IY+Q)	3	*	*	*	*o	*	1	'FDBEQ'	19Ø	Y
CP K	2	*	*	*	*o	*	1	'FEK'	254	RETURN
CPD	2	–	*	*	*	*	1	'EDA9'	169	D

Instrução	Número de bytes	Flags						Código Hexadecimal	Código Decimal	Caracteres
		C	Z	S	P/O	AC	N			
CPDR	2	–	*	*	*	*	1	'EDB9'	185	T
CPI	2	–	*	*	*	*	1	'EDA1'	161	5
CPIR	2	–	*	*	*	*	1	'EDB1'	177	L

E) **INSTRUÇÕES LÓGICAS E OPERAÇÕES COM BITS; INSTRUÇÕES DE "ROTAÇÃO"**

Notação genérica:

AND r
AND (HXY+Q)
AND K
OR r
OR (HXY+Q)
OR K
XOR r
XOR (HXY+Q)
XOR K
BIT b,r
BIT b,(HXY+Q)
SET b,r
SET b,(HXY+Q)
RES b,r
RES b,(HXY+Q)
RLC r
RLC (HXY+Q)
RRC r
RRC (HXY+Q)
RL r
RL (HXY+Q)
RR r
RR (HXY+Q)
SLA r
SLA (HXY+Q)
SRA r
SRA (HXY+Q)
SRL r
SRL (HXY+Q)
RLCA
RRCA
RLA
RRA
RLD
RRD
CPL
CCF
SCF

Observação: (+Q) não vale para HL.

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
AND A	1	Ø	*	*	*p	1	Ø	'A7'	167	[B]
AND B	1	Ø	*	*	*p	1	Ø	'AØ'	16Ø	[4]
AND C	1	Ø	*	*	*p	1	Ø	'A1'	161	[5]
AND D	1	Ø	*	*	*p	1	Ø	'A2'	162	[6]
AND E	1	Ø	*	*	*p	1	Ø	'A3'	163	[7]
AND H	1	Ø	*	*	*p	1	Ø	'A4'	164	[8]
AND L	1	Ø	*	*	*p	1	Ø	'A5'	165	[9]
AND (HL)	1	Ø	*	*	*p	1	Ø	'A6'	166	[A]
AND (IX+Q)	3	Ø	*	*	*p	1	Ø	'DDA6Q'	166	[A]
AND (IY+Q)	3	Ø	*	*	*p	1	Ø	'FDA6Q'	166	[A]
AND K	2	Ø	*	*	*p	1	Ø	'E6K'	23Ø	NEW
OR A	1	Ø	*	*	*p	1	Ø	'B7'	183	[R]
OR B	1	Ø	*	*	*p	1	Ø	'BØ'	176	[K]
OR C	1	Ø	*	*	*p	1	Ø	'B1'	177	[L]
OR D	1	Ø	*	*	*p	1	Ø	'B2'	178	[M]
OR E	1	Ø	*	*	*p	1	Ø	'B3'	179	[N]
OR H	1	Ø	*	*	*p	1	Ø	'B4'	18Ø	[O]
OR L	1	Ø	*	*	*p	1	Ø	'B5'	181	[P]
OR (HL)	1	Ø	*	*	*p	1	Ø	'B6'	182	[Q]
OR (IX+Q)	3	Ø	*	*	*p	1	Ø	'DDB6Q'	182	[Q]
OR (IY+Q)	3	Ø	*	*	*p	1	Ø	'FDB6Q'	182	[Q]
OR K	2	Ø	*	*	*p	1	Ø	'F6K'	246	PLOT
XOR A	1	Ø	*	*	*p	1	Ø	'AF'	175	[J]
XOR B	1	Ø	*	*	*p	1	Ø	'A8'	168	[C]
XOR C	1	Ø	*	*	*p	1	Ø	'A9'	169	[D]
XOR D	1	Ø	*	*	*p	1	Ø	'AA'	17Ø	[E]
XOR E	1	Ø	*	*	*p	1	Ø	'AB'	171	[F]
XOR H	1	Ø	*	*	*p	1	Ø	'AC'	172	[G]
XOR L	1	Ø	*	*	*p	1	Ø	'AD'	173	[H]

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
XOR (HL)	1	Ø	*	*	*p	1	Ø	'AE'	174	[I]
XOR (IX+Q)	3	Ø	*	*	*p	1	Ø	'DDAEQ'	174	[I]
XOR (IY+Q)	3	Ø	*	*	*p	1	Ø	'FDAEQ'	174	[I]
XOR K	2	Ø	*	*	*p	1	Ø	'EEK'	238	INPUT
BIT Ø,A	2	–	*	?	?	1	Ø	'CB47'	71	
BIT Ø,B	2	–	*	?	?	1	Ø	'CB4Ø'	64	RND
BIT Ø,C	2	–	*	?	?	1	Ø	'CB41'	65	INKEY$
BIT Ø,D	2	–	*	?	?	1	Ø	'CB42'	66	PI
BIT Ø,E	2	–	*	?	?	1	Ø	'CB43'	67	
BIT Ø,H	2	–	*	?	?	1	Ø	'CB44'	68	
BIT Ø,L	2	–	*	?	?	1	Ø	'CB45'	69	
BIT 1,A	2	–	*	?	?	1	Ø	'CB4F'	79	
BIT 1,B	2	–	*	?	?	1	Ø	'CB48'	72	
BIT 1,C	2	–	*	?	?	1	Ø	'CB49'	73	
BIT 1,D	2	–	*	?	?	1	Ø	'CB4A'	74	
BIT 1,E	2	–	*	?	?	1	Ø	'CB4B'	75	
BIT 1,H	2	–	*	?	?	1	Ø	'CB4C'	76	
BIT 1,L	2	–	*	?	?	1	Ø	'CB4D'	77	
BIT 2,A	2	–	*	?	?	1	Ø	'CB57'	87	
BIT 2,B	2	–	*	?	?	1	Ø	'CB5Ø'	8Ø	
BIT 2,C	2	–	*	?	?	1	Ø	'CB51'	81	
BIT 2,D	2	–	*	?	?	1	Ø	'CB52'	82	
BIT 2,E	2	–	*	?	?	1	Ø	'CB53'	83	
BIT 2,H	2	–	*	?	?	1	Ø	'CB54'	84	
BIT 2,L	2	–	*	?	?	1	Ø	'CB55'	85	
BIT 3,A	2	–	*	?	?	1	Ø	'CB5F'	95	
BIT 3,B	2	–	*	?	?	1	Ø	'CB58'	88	
BIT 3,C	2	–	*	?	?	1	Ø	'CB59'	89	
BIT 3,D	2	–	*	?	?	1	Ø	'CB5A'	9Ø	

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
BIT 3,E	2	–	*	?	?	1	Ø	'CB5B'	91	
BIT 3,H	2	–	*	?	?	1	Ø	'CB5C'	92	
BIT 3,L	2	–	*	?	?	1	Ø	'CB5D'	93	
BIT 4,A	2	–	*	?	?	1	Ø	'CB67'	1Ø3	
BIT 4,B	2	–	*	?	?	1	Ø	'CB6Ø'	96	
BIT 4,C	2	–	*	?	?	1	Ø	'CB61'	97	
BIT 4,D	2	–	*	?	?	1	Ø	'CB62'	98	
BIT 4,E	2	–	*	?	?	1	Ø	'CB63'	99	
BIT 4,H	2	–	*	?	?	1	Ø	'CB64'	1ØØ	
BIT 4,L	2	–	*	?	?	1	Ø	'CB65'	1Ø1	
BIT 5,A	2	–	*	?	?	1	Ø	'CB6F'	111	
BIT 5,B	2	–	*	?	?	1	Ø	'CB68'	1Ø4	
BIT 5,C	2	–	*	?	?	1	Ø	'CB69'	1Ø5	
BIT 5,D	2	–	*	?	?	1	Ø	'CB6A'	1Ø6	
BIT 5,E	2	–	*	?	?	1	Ø	'CB6B'	1Ø7	
BIT 5,H	2	–	*	?	?	1	Ø	'CB6C'	1Ø8	
BIT 5,L	2	–	*	?	?	1	Ø	'CB6D'	1Ø9	
BIT 6,A	2	–	*	?	?	1	Ø	'CB77'	119	RUBOUT
BIT 6,B	2	–	*	?	?	1	Ø	'CB7Ø'	112	⇧
BIT 6,C	2	–	*	?	?	1	Ø	'CB71'	113	⇩
BIT 6,D	2	–	*	?	?	1	Ø	'CB72'	114	⇦
BIT 6,E	2	–	*	?	?	1	Ø	'CB73'	115	⇨
BIT 6,H	2	–	*	?	?	1	Ø	'CB74'	116	GRAPHICS
BIT 6,L	2	–	*	?	?	1	Ø	'CB75'	117	EDIT
BIT 7,A	2	–	*	?	?	1	Ø	'CB7F'	127	CURSOR
BIT 7,B	2	–	*	?	?	1	Ø	'CB78'	12Ø	MODO
BIT 7,C	2	–	*	?	?	1	Ø	'CB79'	121	FUNCTION
BIT 7,D	2	–	*	?	?	1	Ø	'CB7A'	122	
BIT 7,E	2	–	*	?	?	1	Ø	'CB7B'	123	

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
BIT 7,H	2	–	*	?	?	1	Ø	'CB7C'	124	
BIT 7,L	2	–	*	?	?	1	Ø	'CB7D'	125	
BIT Ø,(HL)	2	–	*	?	?	1	Ø	'CB46'	7Ø	
BIT Ø, (IX+Q)	4	–	*	?	?	1	Ø	'DDCBQ46'	7Ø	
BIT Ø, (IY+Q)	4	–	*	?	?	1	Ø	'FDCBQ46'	7Ø	
BIT 1,(HL)	2	–	*	?	?	1	Ø	'CB4E'	78	
BIT 1,(IX+Q)	4	–	*	?	?	1	Ø	'DDCBQ4E'	78	
BIT 1,(IY+Q)	4	–	*	?	?	1	Ø	'FDCBQ4E'	78	
BIT 2,(HL)	2	–	*	?	?	1	Ø	'CB56'	86	
BIT 2,(IX+Q)	4	–	*	?	?	1	Ø	'DDCBQ56'	86	
BIT 2,(IY+Q)	4	–	*	?	?	1	Ø	'FDCBQ56'	86	
BIT 3,(HL)	2	–	*	?	?	1	Ø	'CB5E'	94	
BIT 3,(IX+Q)	4	–	*	?	?	1	Ø	'DDCBQ5E'	94	
BIT 3,(IY+Q)	4	–	*	?	?	1	Ø	'FDCBQ5E'	94	
BIT 4,(HL)	2	–	*	?	?	1	Ø	'CB66'	1Ø2	
BIT 4,(IX+Q)	4	–	*	?	?	1	Ø	'DDCBQ66'	1Ø2	
BIT 4,(IY+Q)	4	–	*	?	?	1	Ø	'FDCBQ66'	1Ø2	
BIT 5,(HL)	2	–	*	?	?	1	Ø	'CB6E'	11Ø	
BIT 5,(IX+Q)	4	–	*	?	?	1	Ø	'DDCBQ6E'	11Ø	
BIT 5,(IY+Q)	4	–	*	?	?	1	Ø	'FDCBQ6E'	11Ø	
BIT 6,(HL)	2	–	*	?	?	1	Ø	'CB76'	118	NEW LINE
BIT 6,(IX+Q)	4	–	*	?	?	1	Ø	'DDCBQ76'	118	NEW LINE
BIT 6,(IY+Q)	4	–	*	?	?	1	Ø	'FDCBQ76'	118	NEW LINE
BIT 7,(HL)	2	–	*	?	?	1	Ø	'CB7E'	126	número
BIT 7,(IX+Q)	4	–	*	?	?	1	Ø	'DDCBQ7E'	126	número
BIT 7,(IY+Q)	4	–	*	?	?	1	Ø	'FDCBQ7E'	126	número
SET Ø,A	2	–	–	–	–	–	–	'CBC7'	199	SIN
SET Ø,B	2	–	–	–	–	–	–	'CBCØ'	192	" "

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
SET Ø,C	2	–	–	–	–	–	–	'CBC1'	193	AT
SET Ø,D	2	–	–	–	–	–	–	'CBC2'	194	TAB
SET Ø,E	2	–	–	–	–	–	–	'CBC3'	195	
SET Ø,H	2	–	–	–	–	–	–	'CBC4'	196	CODE
SET Ø,L	2	–	–	–	–	–	–	'CBC5'	197	VAL
SET 1,A	2	–	–	–	–	–	–	'CBCF'	2Ø7	INT
SET 1,B	2	–	–	–	–	–	–	'CBC8'	2ØØ	COS
SET 1,C	2	–	–	–	–	–	–	'CBC9'	2Ø1	TAN
SET 1,D	2	–	–	–	–	–	–	'CBCA'	2Ø2	ASN
SET 1,E	2	–	–	–	–	–	–	'CBCB'	2Ø3	ACS
SET 1,H	2	–	–	–	–	–	–	'CBCC'	2Ø4	ATN
SET 1,L	2	–	–	–	–	–	–	'CBCD'	2Ø5	LN
SET 2,A	2	–	–	–	–	–	–	'CBD7'	215	NOT
SET 2,B	2	–	–	–	–	–	–	'CBDØ'	2Ø8	SQR
SET 2,C	2	–	–	–	–	–	–	'CBD1'	2Ø9	SGN
SET 2,D	2	–	–	–	–	–	–	'CBD2'	21Ø	ABS
SET 2,E	2	–	–	–	–	–	–	'CBD3'	211	PEEK
SET 2,H	2	–	–	–	–	–	–	'CBD4'	212	USR
SET 2,L	2	–	–	–	–	–	–	'CBD5'	213	STR$
SET 3,A	2	–	–	–	–	–	–	'CBDF'	223	TO
SET 3,B	2	–	–	–	–	–	–	'CBD8'	216	**
SET 3,C	2	–	–	–	–	–	–	'CBD9'	217	OR
SET 3,D	2	–	–	–	–	–	–	'CBDA'	218	AND
SET 3,E	2	–	–	–	–	–	–	'CBDB'	219	<=
SET 3,H	2	–	–	–	–	–	–	'CBDC'	22Ø	>=
SET 3,L	2	–	–	–	–	–	–	'CBDD'	221	<>
SET 4,A	2	–	–	–	–	–	–	'CBE7'	231	SCROLL
SET 4,B	2	–	–	–	–	–	–	'CBEØ'	224	STEP
SET 4,C	2	–	–	–	–	–	–	'CBE1'	225	LPRINT

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
SET 4,D	2	–	–	–	–	–	–	'CBE2'	226	LLIST
SET 4,E	2	–	–	–	–	–	–	'CBE3'	227	STOP
SET 4,H	2	–	–	–	–	–	–	'CBE4'	228	SLOW
SET 4,L	2	–	–	–	–	–	–	'CBE5'	229	FAST
SET 5,A	2	–	–	–	–	–	–	'CBEF'	239	LOAD
SET 5,B	2	–	–	–	–	–	–	'CBE8'	232	CONT
SET 5,C	2	–	–	–	–	–	–	'CBE9'	233	DIM
SET 5,D	2	–	–	–	–	–	–	'CBEA'	234	REM
SET 5,E	2	–	–	–	–	–	–	'CBEB'	235	FOR
SET 5,H	2	–	–	–	–	–	–	'CBEC'	236	GOTO
SET 5,L	2	–	–	–	–	–	–	'CBED'	237	GOSUB
SET 6,A	2	–	–	–	–	–	–	'CBF7'	247	RUN
SET 6,B	2	–	–	–	–	–	–	'CBFØ'	24Ø	LIST
SET 6,C	2	–	–	–	–	–	–	'CBF1'	241	LET
SET 6,D	2	–	–	–	–	–	–	'CBF2'	242	PAUSE
SET 6,E	2	–	–	–	–	–	–	'CBF3'	243	NEXT
SET 6,H	2	–	–	–	–	–	–	'CBF4'	244	POKE
SET 6,L	2	–	–	–	–	–	–	'CBF5'	245	PRINT
SET 7,A	2	–	–	–	–	–	–	'CBFF'	255	COPY
SET 7,B	2	–	–	–	–	–	–	'CBF8'	248	SAVE
SET 7,C	2	–	–	–	–	–	–	'CBF9'	249	RAND
SET 7,D	2	–	–	–	–	–	–	'CBFA'	25Ø	IF
SET 7,E	2	–	–	–	–	–	–	'CBFB'	251	CLS
SET 7,H	2	–	–	–	–	–	–	'CBFC'	252	UNPLOT
SET 7,L	2	–	–	–	–	–	–	'CBFD'	253	CLEAR
SET Ø,(HL)	2	–	–	–	–	–	–	'CBC6'	198	LEN
SET Ø,(IX+Q)	4	–	–	–	–	–	–	'DDCBQC6'	198	LEN
SET Ø,(IY+Q)	4	–	–	–	–	–	–	'FDCBQC6'	198	LEN
SET 1,(HL)	2	–	–	–	–	–	–	'CBCE'	2Ø6	EXP

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
SET 1,(IX+Q)	4	—	—	—	—	—	—	'DDCBQCE'	2Ø6	EXP
SET 1,(IY+Q)	4	—	—	—	—	—	—	'FDCBQCE'	2Ø6	EXP
SET 2,(HL)	2	—	—	—	—	—	—	'CBD6'	214	CHR$
SET 2,(IX+Q)	4	—	—	—	—	—	—	'DDCBQD6'	214	CHR$
SET 2,(IY+Q)	4	—	—	—	—	—	—	'FDCBQD6'	214	CHR$
SET 3,(HL)	2	—	—	—	—	—	—	'CBDE'	222	THEN
SET 3,(IX+Q)	4	—	—	—	—	—	—	'DDCBQDE'	222	THEN
SET 3,(IY+Q)	4	—	—	—	—	—	—	'FDCBQDE'	222	THEN
SET 4,(HL)	2	—	—	—	—	—	—	'CBE6'	23Ø	NEW
SET 4,(IX+Q)	4	—	—	—	—	—	—	'DDCBQE6'	23Ø	NEW
SET 4,(IY+Q)	4	—	—	—	—	—	—	'FDCBQE6'	23Ø	NEW
SET 5,(HL)	2	—	—	—	—	—	—	'CBEE'	238	INPUT
SET 5,(IX+Q)	4	—	—	—	—	—	—	'DDCBQEE'	238	INPUT
SET 5,(IY+Q)	4	—	—	—	—	—	—	'FDCBQEE'	238	INPUT
SET 6,(HL)	2	—	—	—	—	—	—	'CBF6'	246	PLOT
SET 6,(IX+Q)	4	—	—	—	—	—	—	'DDCBQF6'	246	PLOT
SET 6,(IY+Q)	4	—	—	—	—	—	—	'FDCBQF6'	246	PLOT
SET 7,(HL)	2	—	—	—	—	—	—	'CBFE'	254	RETURN
SET 7,(IX+Q)	4	—	—	—	—	—	—	'DDCBQFE'	254	RETURN
SET 7,(IY+Q)	4	—	—	—	—	—	—	'FDCBQFE'	254	RETURN
RES Ø,A	2	—	—	—	—	—	—	'CB87'	135	
RES Ø,B	2	—	—	—	—	—	—	'CB8Ø'	128	
RES Ø,C	2	—	—	—	—	—	—	'CB81'	129	
RES Ø,D	2	—	—	—	—	—	—	'CB82'	13Ø	
RES Ø,E	2	—	—	—	—	—	—	'CB83'	131	
RES Ø,H	2	—	—	—	—	—	—	'CB84'	132	
RES Ø,L	2	—	—	—	—	—	—	'CB85'	133	
RES 1,A	2	—	—	—	—	—	—	'CB8F'	143	?
RES 1,B	2	—	—	—	—	—	—	'CB88'	136	

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
RES 1,C	2	—	—	—	—	—	—	'CB89'	137	
RES 1,D	2	—	—	—	—	—	—	'CB8A'	138	
RES 1,E	2	—	—	—	—	—	—	'CB8B'	139	"
RES 1,H	2	—	—	—	—	—	—	'CB8C'	14Ø	£
RES 1,L	2	—	—	—	—	—	—	'CB8D'	141	$
RES 2,A	2	—	—	—	—	—	—	'CB97'	151	*
RES 2,B	2	—	—	—	—	—	—	'CB9Ø'	144	(
RES 2,C	2	—	—	—	—	—	—	'CB91'	145	)
RES 2,D	2	—	—	—	—	—	—	'CB92'	146	>
RES 2,E	2	—	—	—	—	—	—	'CB93'	147	<
RES 2,H	2	—	—	—	—	—	—	'CB94'	148	=
RES 2,L	2	—	—	—	—	—	—	'CB95'	149	+
RES 3,A	2	—	—	—	—	—	—	'CB9F'	159	3
RES 3,B	2	—	—	—	—	—	—	'CB98'	152	/
RES 3,C	2	—	—	—	—	—	—	'CB99'	153	;
RES 3,D	2	—	—	—	—	—	—	'CB9A'	154	,
RES 3,E	2	—	—	—	—	—	—	'CB9B'	155	.
RES 3,H	2	—	—	—	—	—	—	'CB9C'	156	Ø
RES 3,L	2	—	—	—	—	—	—	'CB9D'	157	1
RES 4,A	2	—	—	—	—	—	—	'CBA7'	167	B
RES 4,B	2	—	—	—	—	—	—	'CBAØ'	16Ø	4
RES 4,C	2	—	—	—	—	—	—	'CBA1'	161	5
RES 4,D	2	—	—	—	—	—	—	'CBA2'	162	6
RES 4,E	2	—	—	—	—	—	—	'CBA3'	163	7
RES 4,H	2	—	—	—	—	—	—	'CBA4'	164	8
RES 4,L	2	—	—	—	—	—	—	'CBA5'	165	9
RES 5,A	2	—	—	—	—	—	—	'CBAF'	175	J
RES 5,B	2	—	—	—	—	—	—	'CBA8'	168	C
RES 5,C	2	—	—	—	—	—	—	'CBA9'	169	D

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
RES 5,D	2	–	–	–	–	–	–	'CBAA'	17Ø	E
RES 5,E	2	–	–	–	–	–	–	'CBAB'	171	F
RES 5,H	2	–	–	–	–	–	–	'CBAC'	172	G
RES 5,L	2	–	–	–	–	–	–	'CBAD'	173	H
RES 6,A	2	–	–	–	–	–	–	'CBB7'	183	R
RES 6,B	2	–	–	–	–	–	–	'CBBØ'	176	K
RES 6,C	2	–	–	–	–	–	–	'CBB1'	177	L
RES 6,D	2	–	–	–	–	–	–	'CBB2'	178	M
RES 6,E	2	–	–	–	–	–	–	'CBB3'	179	N
RES 6,H	2	–	–	–	–	–	–	'CBB4'	18Ø	O
RES 6,L	2	–	–	–	–	–	–	'CBB5'	181	P
RES 7,A	2	–	–	–	–	–	–	'CBBF'	191	Z
RES 7,B	2	–	–	–	–	–	–	'CBB8'	184	S
RES 7,C	2	–	–	–	–	–	–	'CBB9'	185	T
RES 7,D	2	–	–	–	–	–	–	'CBBA'	186	U
RES 7,E	2	–	–	–	–	–	–	'CBBB'	187	V
RES 7,H	2	–	–	–	–	–	–	'CBBC'	188	W
RES 7,L	2	–	–	–	–	–	–	'CBBD'	189	X
RES Ø,(HL)	2	–	–	–	–	–	–	'CB86'	134	
RES Ø,(IX+Q)	4	–	–	–	–	–	–	'DDCBQ86'	134	
RES Ø,(IY+Q)	4	–	–	–	–	–	–	'FDCBQ86	134	
RES 1,(HL)	2	–	–	–	–	–	–	'CB8E'	142	:
RES 1,(IX+Q)	4	–	–	–	–	–	–	'DDCBQ8E'	142	:
RES 1,(IY+Q)	4	–	–	–	–	–	–	'FDCBQ8E'	142	:
RES 2,(HL)	2	–	–	–	–	–	–	'CB96'	15Ø	–
RES 2,(IX+Q)	4	–	–	–	–	–	–	'DDCBQ96'	15Ø	–
RES 2,(IY+Q)	4	–	–	–	–	–	–	'FDCBQ96'	15Ø	–
RES 3,(HL)	2	–	–	–	–	–	–	'CB9E'	158	2
RES 3,(IX+Q)	4	–	–	–	–	–	–	'DDCBQ9E'	158	2

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
RES 3,(IY+Q)	4	–	–	–	–	–	–	'FDCBQ9E'	158	2
RES 4,(HL)	2	–	–	–	–	–	–	'CBA6'	166	A
RES 4,(IX+Q)	4	–	–	–	–	–	–	'DDCBQA6'	166	A
RES 4,(IY+Q)	4	–	–	–	–	–	–	'FDCBQA6'	166	A
RES 5,(HL)	2	–	–	–	–	–	–	'CBAE'	174	I
RES 5,(IX+Q)	4	–	–	–	–	–	–	'DDCBQAE'	174	I
RES 5,(IY+Q)	4	–	–	–	–	–	–	'FDCBQAE'	174	I
RES 6,(HL)	2	–	–	–	–	–	–	'CBB6'	182	Q
RES 6,(IX+Q)	4	–	–	–	–	–	–	'DDCBQB6'	182	Q
RES 6,(IY+Q)	4	–	–	–	–	–	–	'FDCBQB6'	182	Q
RES 7,(HL)	2	–	–	–	–	–	–	'CBBE'	19Ø	Y
RES 7,(IX+Q)	4	–	–	–	–	–	–	'DDCBQBE'	19Ø	Y
RES 7,(IY+Q)	4	–	–	–	–	–	–	'FDCBQBE'	19Ø	Y
RLC A	2	*	*	*	*p	Ø	Ø	'CBØ7'	7	
RLC B	2	*	*	*	*p	Ø	Ø	'CBØØ'	Ø	
RLC C	2	*	*	*	*p	Ø	Ø	'CBØ1'	1	
RLC D	2	*	*	*	*p	Ø	Ø	'CBØ2'	2	
RLC E	2	*	*	*	*p	Ø	Ø	'CBØ3'	3	
RLC H	2	*	*	*	*p	Ø	Ø	'CBØ4'	4	
RLC L	2	*	*	*	*p	Ø	Ø	'CBØ5'	5	
RLC (HL)	2	*	*	*	*p	Ø	Ø	'CBØ6'	6	
RLC (IX+Q)	4	*	*	*	*p	Ø	Ø	'DDCBQØ6'	6	
RLC (IY+Q)	4	*	*	*	*p	Ø	Ø	'FDCBQØ6'	6	
RRC A	2	*	*	*	*p	Ø	Ø	'CBØF'	15	?
RRC B	2	*	*	*	*p	Ø	Ø	'CBØ8'	8	
RRC C	2	*	*	*	*p	Ø	Ø	'CBØ9'	9	
RRC D	2	*	*	*	*p	Ø	Ø	'CBØA'	1Ø	
RRC E	2	*	*	*	*p	Ø	Ø	'CBØB'	11	"
RRC H	2	*	*	*	*p	Ø	Ø	'CBØC'	12	£

Instrução	Número de bytes	Flags						Código Hexadecimal	Código Decimal	Caracteres
		C	Z	S	P/O	AC	N			
RRC L	2	*	*	*	*p	Ø	Ø	'CBØD'	13	$
RRC (HL)	2	*	*	*	*p	Ø	Ø	'CBØE'	14	:
RRC (IX+Q)	4	*	*	*	*p	Ø	Ø	'DDCBQØE'	14	:
RRC (IY+Q)	4	*	*	*	*p	Ø	Ø	'FDCBQØE'	14	:
RL A	2	*	*	*	*p	Ø	Ø	'CB17'	23	*
RL B	2	*	*	*	*p	Ø	Ø	'CB1Ø'	16	(
RL C	2	*	*	*	*p	Ø	Ø	'CB11'	17	)
RL D	2	*	*	*	*p	Ø	Ø	'CB12'	18	>
RL E	2	*	*	*	*p	Ø	Ø	'CB13'	19	<
RL H	2	*	*	*	*p	Ø	Ø	'CB14'	2Ø	=
RL L	2	*	*	*	*p	Ø	Ø	'CB15'	21	+
RL (HL)	2	*	*	*	*p	Ø	Ø	'CB16'	22	–
RL (IX+Q)	4	*	*	*	*p	Ø	Ø	'DDCBQ16'	22	–
RL (IY+Q)	4	*	*	*	*p	Ø	Ø	'FDCBQ16'	22	–
RR A	2	*	*	*	*p	Ø	Ø	'CB1F'	31	3
RR B	2	*	*	*	*p	Ø	Ø	'CB18'	24	/
RR C	2	*	*	*	*p	Ø	Ø	'CB19'	25	;
RR D	2	*	*	*	*p	Ø	Ø	'CB1A'	26	,
RR E	2	*	*	*	*p	Ø	Ø	'CB1B'	27	.
RR H	2	*	*	*	*p	Ø	Ø	'CB1C'	28	Ø
RR L	2	*	*	*	*p	Ø	Ø	'CB1D'	29	1
RR (HL)	2	*	*	*	*p	Ø	Ø	'CB1E'	3Ø	2
RR (IX+Q)	4	*	*	*	*p	Ø	Ø	'DDCBQ1E'	3Ø	2
RR (IY+Q)	4	*	*	*	*p	Ø	Ø	'FDCBQ1E'	3Ø	2
SLA A	2	*	*	*	*p	Ø	Ø	'CB27'	39	B
SLA B	2	*	*	*	*p	Ø	Ø	'CB2Ø'	32	4
SLA C	2	*	*	*	*p	Ø	Ø	'CB21'	33	5
SLA D	2	*	*	*	*p	Ø	Ø	'CB22'	34	6
SLA E	2	*	*	*	*p	Ø	Ø	'CB23'	35	7

Instrução	Número de bytes	Flags						Código Hexadecimal	Código Decimal	Caracteres
		C	Z	S	P/O	AC	N			
SLA H	2	*	*	*	*p	Ø	Ø	'CB24'	36	8
SLA L	2	*	*	*	*p	Ø	Ø	'CB25'	37	9
SLA (HL)	2	*	*	*	*p	Ø	Ø	'CB26'	38	A
SLA (IX+Q)	4	*	*	*	*p	Ø	Ø	'DDCBQ26'	38	A
SLA (IY+Q)	4	*	*	*	*p	Ø	Ø	'FDCBQ26'	38	A
SRA A	2	*	*	*	*p	Ø	Ø	'CB2F'	47	J
SRA B	2	*	*	*	*p	Ø	Ø	'CB28'	4Ø	C
SRA C	2	*	*	*	*p	Ø	Ø	'CB29'	41	D
SRA D	2	*	*	*	*p	Ø	Ø	'CB2A'	42	E
SRA E	2	*	*	*	*p	Ø	Ø	'CB2B'	43	F
SRA H	2	*	*	*	*p	Ø	Ø	'CB2C'	44	G
SRA L	2	*	*	*	*p	Ø	Ø	'CB2D'	45	H
SRA (HL)	2	*	*	*	*p	Ø	Ø	'CB2E'	46	I
SRA (IX+Q)	4	*	*	*	*p	Ø	Ø	'DDCBQ2E'	46	I
SRA (IY+Q)	4	*	*	*	*p	Ø	Ø	'FDCBQ2E'	46	I
SRL A	2	*	*	*	*p	Ø	Ø	'CB3F'	63	Z
SRL B	2	*	*	*	*p	Ø	Ø	'CB38'	56	S
SRL C	2	*	*	*	*p	Ø	Ø	'CB39'	57	T
SRL D	2	*	*	*	*p	Ø	Ø	'CB3A'	58	U
SRL E	2	*	*	*	*p	Ø	Ø	'CB3B'	59	V
SRL H	2	*	*	*	*p	Ø	Ø	'CB3C'	6Ø	W
SRL L	2	*	*	*	*p	Ø	Ø	'CB3D'	61	X
SRL (HL)	2	*	*	*	*p	Ø	Ø	'CB3E'	62	Y
SRL (IX+Q)	4	*	*	*	*p	Ø	Ø	'DDCBQ3E'	62	Y
SRL (IY+Q)	4	*	*	*	*p	Ø	Ø	'FDCBQ3E'	62	Y
RLCA	1	*	–	–	–	Ø	Ø	'Ø7'	7	▛
RRCA	1	*	–	–	–	Ø	Ø	'ØF'	15	?
RLA	1	*	–	–	–	Ø	Ø	'17'	23	*
RRA	1	*	–	–	–	Ø	Ø	'1F'	31	3

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
RLD	2	–	*	*	*p	Ø	Ø	'ED6F'	111	
RRD	2	–	*	*	*p	Ø	Ø	'ED67'	1Ø3	
CPL	1	–	–	–	–	1	1	'2F'	47	J
CCF	1	*	–	–	–	?	Ø	'3F'	63	Z
SCF	1	1	–	–	–	Ø	Ø	'37'	55	R

F) **INSTRUÇÕES DE INTERRUPÇÃO – INSTRUÇÕES DE ENTRADA E SAÍDA**

Notação genérica:

EI
DI
IM { Ø, 1, 2 }
RETI
RETN
RST { Ø, 8, 16, 24, 32, 4Ø, 48, 56 }
IN A, K
IN r, (C)
IN (repetição)
OUT K,A
OUT (C),r
OUT (repetição)
LD A,I
LD I,A

Instrução	Número de bytes	Flags C	Z	S	P/O	AC	N	Código Hexadecimal	Código Decimal	Caracteres
EI	1	–	–	–	–	–	–	'FB'	251	CLS
DI	1	–	–	–	–	–	–	'F3'	243	NEXT
IM Ø	2	–	–	–	–	–	–	'ED46'	7Ø	
IM 1	2	–	–	–	–	–	–	'ED56'	86	
IM 2	2	–	–	–	–	–	–	'ED5E'	94	
RETI	2	–	–	–	–	–	–	'ED4D'	77	
RETN	2	–	–	–	–	–	–	'ED45'	69	
RST Ø	1	–	–	–	–	–	–	'C7'	199	SIN
RST 8	1	–	–	–	–	–	–	'CF'	2Ø7	INT
RST 16	1	–	–	–	–	–	–	'D7'	215	NOT
RST 24	1	–	–	–	–	–	–	'DF'	223	TO
RST 32	1	–	–	–	–	–	–	'E7'	231	SCROLL
RST 4Ø	1	–	–	–	–	–	–	'EF'	239	LOAD
RST 48	1	–	–	–	–	–	–	'F7'	247	RUN
RST 56	1	–	–	–	–	–	–	'FF'	255	COPY
IN A,K	2	–	–	–	–	–	–	'DBK'	219	<=
IN A,(C)	2	–	*	*	*p	*	Ø	'ED78'	12Ø	MODO
IN B,(C)	2	–	*	*	*p	*	Ø	'ED4Ø'	64	RND
IN C,(C)	2	–	*	*	*p	*	Ø	'ED48'	72	
IN D, (C)	2	–	*	*	*p	*	Ø	'ED5Ø'	8Ø	
IN E,(C)	2	–	*	*	*p	*	Ø	'ED58'	88	
IN H,(C)	2	–	*	*	*p	*	Ø	'ED6Ø'	96	
IN L,(C)	2	–	*	*	*p	*	Ø	'ED68'	1Ø4	
IND	2	–	*	?	?	?	1	'EDAA'	17Ø	E
INDR	2	–	1	?	?	?	1	'EDBA'	186	U
INI	2	–	*	?	?	?	1	'EDA2'	162	6
INIR	2	–	1	?	?	?	1	'EDB2'	178	M
OUT K,A	1	–	–	–	–	–	–	'D3K'	211	PEEK
OUT (C),A	2	–	–	–	–	–	–	'ED79'	121	FUNCTION

Instrução	Número de bytes	Flags						Código Hexadecimal	Código Decimal	Caracteres
		C	Z	S	P/O	AC	N			
OUT (C),B	2	–	–	–	–	–	–	'ED41'	65	INKEY$
OUT (C),C	2	–	–	–	–	–	–	'ED49'	73	
OUT (C),D	2	–	–	–	–	–	–	'ED51'	81	
OUT (C),E	2	–	–	–	–	–	–	'ED59'	89	
OUT (C),H	2	–	–	–	–	–	–	'ED61'	97	
OUT (C),L	2	–	–	–	–	–	–	'ED69'	1Ø5	
OUTD	2	–	*	?	?	?	1	'EDAB'	171	F
OTDR	2	–	1	?	?	?	1	'EDBB'	187	V
OUTI	2	–	*	?	?	?	1	'EDA3'	163	7
OTIR	2	–	1	?	?	?	1	'EDB3'	179	N
LD A,I	2	–	*	*	T	Ø	Ø	'ED57'	87	
LD I,A	2	–	–	–	–	–	–	'ED47'	71	

G) **OUTRAS INSTRUÇÕES**

Instrução	Número de bytes	Flags						Código Hexadecimal	Código Decimal	Caracteres
		C	Z	S	P/O	AC	N			
EX AF,A'F'	1	–	–	–	–	–	–	'Ø8'	8	□
EXX	1	–	–	–	–	–	–	'D9'	217	OR
EX DE,HL	1	–	–	–	–	–	–	'EB'	235	FOR
EX (SP),HL	1	–	–	–	–	–	–	'E3'	227	STOP
EX (SP),IX	2	–	–	–	–	–	–	'DDE3'	227	STOP
EX (SP),IY	2	–	–	–	–	–	–	'FDE3'	227	STOP
NOP	1	–	–	–	–	–	–	'ØØ'	Ø	□
HALT	1	–	–	–	–	–	–	'76'	118	NEW LINE
DAA	1	*	*	*	*p	*	–	'27'	39	B
RLD	2	–	*	*	*p	Ø	Ø	'ED6F'	111	
RRD	2	–	*	*	*p	Ø	Ø	'ED67'	1Ø3	
LD A,R	2	–	*	*	T	Ø	Ø	'ED5F'	95	
LD R,A	2	–	–	–	–	–	–	'ED4F'	79	

LOVE IS THE ANSWER,
AND YOU KNOW THAT FOR SURE.
LOVE IS A FLOWER, YOU'VE GOTTA
LET IT GROW

LOVE (John Lennon)

Love is real, real is love
Love is feeling, feeling love
Love is wanting to be loved

Love is touch, touch is love
Love is reaching, reaching love
Love is asking to be loved

Love is you, you and me
Love is knowing we can be

Love is free, free is love
Love is living, living love
Love is needing to be loved

JOHN LENNON – 197Ø

ALL YOU NEED IS LOVE (John Lennon & Paul McCartney)

Love, love, love, love, love, love, love, love, love ...

There's noting you can do that can't be done
Nothing you can sing that can't be sung
Nothing you can say but you can learn how to play the game
It's easy...

Nothing you can make that can be made
No one you can save that can't be saved
Nothing you can do but you can learn how to be you in time
It's easy...

BIS { All you need is love, all you need is love
All you need is love, love, love is all you need }

Nothing you can know that isn't known
Nothing you can see that isn't shown
Nowhere you can be that isn't where you're meant to be
It's easy...

BIS { All you need is love, all you need is love
All you need is love, love, love is all you need }

She loves you, yeah, yeah, yeah
She loves you, yeah, yeah, yeah

ALL TOGHETER NOW!
EVERYBODY!

THE BEATLES – 1967

Você sabia que os Beatles tocaram esta música na primeira transmissão *mundial* via satélite?

Bibliografia

Baker, Toni. *Mastering Machine Code on your ZX81.*

Burton, D.P. e Dexter, A.L. *Microprocessor Systems Handbook.*

Lennon, John Ono; McCartney, James Paul; Harrison, George e Starkey, Richard (Ringo Starr). *The Beatles Lyrics.*

Massola, Antonio M.A. e Ferreira, Maria Alice G.V. *A origem do cálculo – estrutura de um computador.*

Osborne, A. *An Introduction to Microcomputers.*

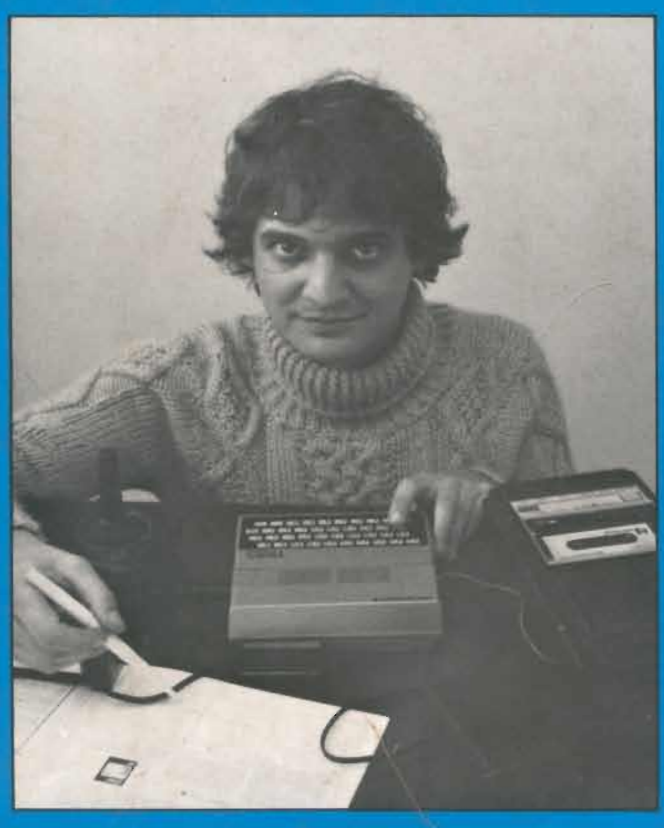
Weller, W.J.; Shatzel, A.V. e Nice, H.Y. *Practical Microcomputer Programming: the INTEL 8080.*

Impresso em offset

GRÁFICA EDITORA HAMBURG

Avenida Bogaert, 64
Vila das Mercês São Paulo
Fone: 914-0233
CEP 04298

com filmes fornecidos pelo editor



FLAVIO ROSSINI,
24 anos, é um exímio contrabaixista e fanático admirador dos Beatles.
Entre uma música e outra formou-se brilhantemente em engenharia
elétrica na USP e está atualmente trabalhando em projetos de
automação industrial, donde sua grande familiaridade
com microprocessadores.
É professor de FÍSICA, BASIC e ASSEMBLY no Núcleo de Orientação de
Estudos em São Paulo e através desta experiência didática motivou-se
a escrever este livro, no qual um assunto árduo como a linguagem de
máquina é esmiuçado de maneira agradável e até humorística.