Este livro, escrito por um jovem de dezasseis anos, foi concebido com o propósito de permitir ao utilizador do ZX Spectrum penetrar no mundo maravilhoso da programação em código-máquina. Partindo dos conceitos básicos desta linguagem, foca os sistemas hexadecimal e binário, os endereços e o modo de atingi-los, os registos, operações aritméticas, e ensina a escrever programas prestando especial atenção aos saltos de execução e às chamadas (instruções CALL, RET, etc.), para além de outros aspectos importantes. A sua parte final é dedicada à teoria dos *stacks*. São apresentados muitos exemplos que facilitam a compreensão dos temas e incluem-se tabelas de grande utilidade.

EDITORIAL PRESENÇA

Código de Máquina para Principiantes

161

JAMES WALSH

CÓDIGO DE MÁQUINA PARA PRINCIPIANTES

ZX SPECTRUM

TEMPOS LIVRES

A. MARQUES

Litec
LIVRARIA EDITORA TÉCNICA LTDA
Rua dos Timbiras 257 - CEP: 01208 - São Paulo
Caixa Postal 30 869 - Tel.: 220-8983
REF 675 PREÇO 95.700

SEBO CULTURA
Compramos, Trocamos e Vendemos
Livros, Revistas, CD'S, LP'S, Fitas
de Vídeo e CD Room (usados).
Rua Benjamin Constant, 1.351
(esquina c/ Hugo Cabral)

CULTURA E TEMPOS LIVRES

1. ABC do Xadrez, *Petar Trifunovitch* e *Sava Vukovitch*
4. ABC do Bridge, *Pierre Jais* e *H. Lahana*
5. Guia Prático de Fotografia, *W. D. Emanuel*
6. ABC do Judo, *E. J. Harrison*
7. Como Fazer Cinema, *Paul Petzold*
8. Bridge Moderno, *Pierre Jais* e *H. Lahana*
9. Fotografia — Técnicas e Truques I, *Edwin Smith*
10. ABC dos Estilos, da Arquitectura ao Mobiliário, *A. Aussel*
11. Fotografia — Técnicas e Truques II, *Edwin Smith*
12. A Pesca Submarina, *António Ribera*
13. Teoria dos Finais de Partida, *Yuri Averbach*
14. Aprenda Rádio, *B. Fighiera*
15. Guia do Cão, *Louise Laliberté-Robert e Jean-Pierre Robert*
16. ABC do Aquário, *Anthony Evans*
17. Iniciação à Electricidade e Electrónica, *Fernand Huré*
18. Os Transístores, *Fernand Huré*
19. Karaté I, *Albrecht Pflüger*
20. Iniciação ao Radiocomando dos Modelos Reduzidos, *C. Péricone*
21. Construa o seu Receptor, *B. Fighiera*
22. Montagens Electrónicas, *B. Fighiera*
23. O Berbequim Eléctrico, *Villy Dreier*
24. Cactos, *J. Nilaus Jensen*
25. Iniciação à Alta Fidelidade, *Peter Turner*
26. O Aquário de Água Doce, *Paulo de Oliveira*
27. ABC do Ténis, *Fonseca Vaz*
28. Karaté II, *Albrecht Pflüger*
29. ABC da Criação de Canários, *Curt Af Enehjelm*
30. Ginástica Feminina, *Sonja Helmer Jensen*
31. Cartomancia, *Thea Koch*
32. Calculadoras Electrónicas de Bolso, *E. Dam Ravn*
33. O Pastor Alemão, *Gilles Legrand*
34. Xadrez — Teoria do Meio Jogo I, *Bondarevsky*
35. Manual do Super 8 — I, *Myron A. Matzkin*
36. ABC da Criação de Periquitos, *Cyril H. Rogers*
37. O Livro dos Gatos, *Bärbel Gerber* e *Horst Bielfeld*
38. Manual do Super 8 — II, *Myron A. Matzkin*
39. ABC do Mergulho Desportivo, *Walter Mattes*
40. Circuitos Integrados/Aplicações Práticas, *F. Bergtold*
41. A Apicultura, *H. R. C. Riches*
42. ABC do Cultivo das Plantas, *H. G. Witham Fogg*
43. ABC da Criação de Pombos, *Kai R. Dahl*
44. Construção de Caixas Acústicas de Alta Fidelidade, *R. Brault*
45. Raças de Canários, *Klaus Speicher*
46. Jogos de Cartas, *Graciano Dolma*
47. Spaniels, *H. S. Lloyd*
48. ABC da Caça, *Fabián Abril*
49. Aprenda Televisão, *Gordon J. King*
50. Iniciação à Pesca, *Juan Nadal*
51. Basquetebol, *Marius Norregard*
52. Cães de Caça, *Santiago Pons*
53. Aprenda Electrónica, *T. L. Squires* e *C. M. Deason*
54. A Avicultura, *Jim Worthington*
55. A Produção de Coelhos, *P. Surdeau* e *R. Henaff*
56. ABC dos Computadores, *T. F. Fry*
57. Natação para crianças, *John Idorn*
58. O Boxer, *Anni Mortensen*
59. Voleibol, *Ole Hansen* e *Per-Göran Person*
60. Iniciação à Vela, *Donald Law*
61. ABC da Filatelia, *Jacqueline Caurat*
62. A Pesca à Beira-Mar, *J.-M. Böelle* e *B. Doyen*
63. Enxerto de Árvores de Fruto, *Alejo Rigau*
64. A Cultura do Morangueiro, *Luís Alsina Grau*
65. Emissores-Receptores (Walkies-Talkies), *P. Duranton*
66. Iniciação à Fotoelectrónica, *Heinz Richter*
67. Doces e Conservas de Fruta, *Robin Howe*
68. A Criação de Hamsters, *C. F. Snow*
69. A Criação de Porcos, *Roy Genders*
70. Calendário do Horticultor, *Luís Alsina Grau*
71. Jogos Electrónicos, *F. G. Rayer*
72. Cultivo de Cogumelos e Trufas, *Alejo Rigau*
73. Aprenda Televisão a Cores, *Gordon J. King*
74. Gravação em Fita Magnética, *Ian R. Sinclair*
75. Poda de Árvores e Arbustos, *Roy Genders*
76. Como Treinar o Seu Cão, *E. Fitch Daglish*
77. Instrumentos de Medida e Verificação, *Heinrich Stöckle*
78. A Criação de Caracóis, *Matías Josa*
79. Rádio — Fundamentos e Técnicas, *Gordon J. King*
80. Como Fazer Gelados, *Sylvie Thiébault*
81. Iniciação à Jardinagem, *Noel Clarasó*
82. A Congelação dos Alimentos, *Suzanne Lapointe*
83. Windfsurf — Prancha à Vela, *Ernstfried Prade*
84. Raças de Cães, *O. Hasselfeldt*
85. Rummy e Canasta, *Claus D. Grupp*
86. A Encadernação, *Annie Persuy*
87. Aprenda Electricidade, *Heinz Richter*
88. Taxidermia, Embalsamamento de Aves e Mamíferos, *Harry Hjortaa*
89. Jogging — Correr para Manter a Forma, *Werner Sonntag*
90. ABC da Cozinha Chinesa, *Sonya Richmond*
91. Jogos T.V., *C. Tavernier*
92. Amplificadores de Som, *Richard Zierl*
93. O Livro do Poker, *Claus D. Grupp*
94. Aprenda a Desenhar, *Rose-Marie de Prémont* e *Nicole Philippi*
95. O Minitrampolim na Escola, *Sonja Helmer Jensen* e *Klaus Dano*
96. Jogos de Luzes e Efeitos Sonoros para Guitarras, *B. Fighiera*
97. O Cultivo do Tomate, *Louis N. Flawn*
98. Pilhas Solares, *F. Juster*
99. A Criação Doméstica de Coelhos, *C. F. Snow*
100. Iniciação ao Futebol, *Wieland Männle* e *Heinz Arnold*
101. Horóscopos Chineses, *Georg Haddenbach*
102. Guia Prático de Marcenaria, *Charles H. Hayward*
103. Andebol, *Fritz* e *Peter Hattig*
104. Dispositivos Anti-Roubo, *H. Schreiber*
105. Perus, Pintadas e Codornizes, *Jérome Sauze*
106. Crepes — Doces e Salgados, *Florence Arzel*
107. Aperitivos e Entradas, *Myrette Tiano*
108. Ténis de Mesa, *Leslie Woollard*
109. Aprenda Surf, *R. Abbott* e *M. Baker*
110. Futebol — Técnica e Táctica, *Kurt Lavall*
111. A Vaca Leiteira, *Colin T. Whittemore*
112. O Cubo Mágico, *Josef Trajber*
113. O Perdigueiro Português, *José M. Correia*
114. Pizzas e Massas à Italiana, *Marieanne Ränk*
115. O Cubo Para Quem Já o Faz, *Josef Trajber*
116. A Pirâmide Mágica, A Torre, O Barril do Diabo, *M. Mrowka-W. J. Weber*
117. Gansos e Patos, *Marie Mourthe*
118. Iniciação ao Kung-Fu, *A. P. Harrington*
119. Electrónica e Fotografia, *Hanns-Peter Siebert*
120. O Livro da Fortuna, *Douglas Hill*
121. Construção de um Alimentador de corrente, *Waldemar Baitinger*
122. Hóquei em Patins, *Francisco Velasco*
123. Técnicas de Tiro, *Anton Kovacic*
124. Aprenda a Tricotar, *Uta Mix*
125. ABC da Patinagem, *Christa-Maria* e *Richard Kerler*
126. A Pesca e os seus Segredos, *Armand Deschamps*
127. O Osciloscópio, *R. Rateau*
128. Guia Prático da Banda do Cidadão, *T. M. Normand*
129. Sumos e Batidos, *Manfred Donderski*
130. Introdução à Programação de Microcomputadores, *Peter C. Sanderson*
131. Aprenda Croché, *Uta Mix*
132. ABC do Microprocessador, *P. Mélusson*
133. Guia Prático de Basic, *Roger Hunt*

134. Introdução à Electrónica Digital, *Ian Sinclair*
135. ABC do Vídeo, *David K. Matthewson*
136. Fotografia em Movimento, *Don Morley*
137. Guia de Cobol, *Ray Welland*
138. Fotografia a Pequena Distância, *Sidney F. Ray*
139. Guia Moderno da Canaricultura, *Manuel Gonçalves*
140. Minielectrónica para Amadores, *Heinz Richter*
141. ABC da Programação de Computadores, *John Shelley*
142. Tarot — O Futuro Pelas Cartas, *Edwin J. Nigg*
143. ABC da Equitação, *Dorothy Johnson*
144. Como Programar o Seu ZX 81, *Patrick Gueulle*
145. 100 Avarias TV e a Maneira Prática de as Detectar, *P. Duranton*
146. ABC da Horticultura, *Louis Giordano*
147. Basic Para Microcomputadores, *A. P. Stephenson*
148. Como Programar o seu ZX Spectrum, *Tim Hartnell* e *Dilwyn Jones*
149. Iniciação aos Motores Diesel, *David S. Maclean*
150. 60 Jogos para o ZX Spectrum, *David Harwood*
151. As Linhas da Mão, *Rosé Hubert*
152. Cozinha Italiana, *Rotraud Degner*
153. Manual do ZX Spectrum, *Simpson* e *Terrel*
154. Z80 Assembler para o ZX Spectrum, *João Paulo Fragoso*
155. Aeróbica, *H. Schulz*
156. ABC do Atletismo, *Denis Watts*
157. 26 Programas Basic para Microcomputadores, *Derrick Daines*
158. Aprenda Pascal no seu Microcomputador, *Jeremy Ruston*
159. Guia Moderno da Suinicultura, *Colin Whittemore*
160. O Bar em Sua Casa — 888 Cocktails, *Aladar von Wesendouk*
161. Código de Máquina para Principiantes, *James Walsh*

JAMES WALSH

CÓDIGO DE MÁQUINA PARA PRINCIPIANTES

EDITORIAL PRESENÇA

Título original:
SPECTRUM MACHINE CODE MADE EASY
VOLUME ONE FOR BEGINNERS
© *Copyright by* James Walsh, 1983
Tradução de Conceição Jardim e Eduardo Nogueira
Capa de Antônio Marques

Reservado todos os direitos
para Portugal à
EDITORIAL PRESENÇA, LDA.
Rua Augusto Gil, 35-A — 1000 LISBOA

DEDICATÓRIA

Para a Margy, com amor
de
nós

A Charles e Emma — OBRIGADO pela ajuda

O AUTOR

James Walsh tinha dezasseis anos quando escreveu este livro, e frequentava nessa altura a escola da Fundação Davenant em Loughton, Essex.

Adquiriu já um bom nome como colaborador de várias revistas de computadores, com distribuição nacional, e contribuiu para a publicação de dois livros sobre computadores Sinclair.

Além de escrever artigos, recensões de software e livros, ainda arranja tempo para ser um fotógrafo apaixonado, um músico medíocre, e é na maioria dos casos uma preocupação para os professores.

Espera publicar um novo livro num futuro próximo — se os exames o permitirem — embora a sua maior ambição seja viajar pelo estrangeiro.

O tempo que lhe foi dado para escrever este livro prejudicou a sua vida social — o que ele espera vir agora a compensar. Entretanto aproveita igualmente para agradecer a paciência e o apoio de toda a sua família e amigos.

INTRODUÇÃO

O objectivo deste livro consiste em fornecer uma introdução completa e digerível à programação em linguagem-máquina — uma introdução que o autor julga necessária porque teria gostado que existisse uma quando começou a interessar-se por este tópico um tanto misterioso. Só mais tarde descobriu que não havia qualquer mistério...

Foram feitos todos os esforços para evitar cair na ratoeira de considerar por um lado qualquer conhecimento prévio da parte do leitor ou de o classificar como um completo idiota! Foi necessário considerar o tema do ponto de vista do "aluno" ou "aluna", tentando sempre prever os problemas que ele ou ela poderiam encontrar. Se o resultado lhe parecer demasiado fácil, talvez este livro não seja para si. Se o considerar muito difícil — peço-lhe desculpa, mas julgo que já não será possível fazer nada por si (tente em vez disto arranjar um lugar de deputado na Assembleia da República...).

OS COMPUTADORES NÃO SABEM FALAR PORTUGUÊS!

I

PARA QUE SERVE O CÓDIGO-MÁQUINA?

A resposta a esta pergunta é muito simplesmente — "O código-máquina é a linguagem própria do computador". É portanto lógica para este. Poderia dar-vos uma definição muito mais complexa e tecnicamente correcta do código-máquina, que no entanto vos deixaria ainda pior informados do que já estão. Este livro serve porém para eliminar o mistério e confusão que rodeia o problema de comunicar com os computadores em geral, particularmente no caso do Spectrum. Foram portanto feitos todos os esforços no sentido de apresentar informações e explicações do modo mais simples possível. Se, depois de ter lido todo o livro, tiver compreendido os aspectos essenciais do código-máquina e tiver uma confiança suficiente em si mesmo para os usar, o esforço terá valido a pena.

O máximo que os computadores conseguem é somar "um mais um", e de facto conseguem chegar a "dois" sem cometerem erros. Isto não é particularmente inteligente, nem aterrorizador. De qualquer modo, estas máquinas parecem ter uma capacidade infinita para amedrontar os seres humanos, de tal modo que estes tendem a convencer-se de que os computadores são muito complicados, muito rápidos, muito inteligentes, ou pelo menos suficientemente terríveis para tentar compreendê-los. Apesar de todas as aparências, o leitor não deverá esquecer porém que o seu Spectrum nada mais é do que uma combinação de componentes electrónicos que, colectivamente, têm capacidade para processarem informações que lhes são dadas de acordo com as indicações que lhes fornecemos. Apesar de o resultado final ter uma aparência sofisticada, trata-se apenas de um instrumento, e de um instrumento sem uma cabeça própria. Apenas pode

fazer o que lhe dizem, e só depois de lhe darmos ordem para tal. Realiza de facto a sua tarefa mais rapidamente e de modo mais eficaz, podendo realizar cálculos bastante complicados numa pequena fracção do tempo de que o cérebro humano necessitaria. No entanto, como já dissémos, a sua única qualidade é ser capaz de calcular rapidamente a soma de um e um.

Qual será o interesse de compreender a natureza de código-máquina e de perder tempo a aprender a utilizá-lo? Não dispomos já de uma "linguagem" perfeitamente adequada para comunicar com o nosso Spectrum — a BASIC? É certo, a BASIC é uma linguagem que o leitor já conhece certamente e que pelo menos parece razoavelmente inteligível. De facto, é precisamente isso que ela é — uma linguagem informática escrita para os seres humanos. O seu objectivo consiste em permitir-nos "falar" com o nosso ZX80/ZX81/Spectrum de uma forma compacta e lógica, usando uma linguagem que compreendemos facilmente, e que pode ser "interpretada" pelo computador. Sim, "interpretada". No interior da memória do computador encontra-se um dicionário completo para tradução de BASIC em código-máquina. O computador baseia-se neste dicionário para interpretar e portanto compreender tudo o que lhe é dito em BASIC, antes de poder realizar qualquer das instruções que lhe são dadas.

Se, portanto, o objectivo é conseguir uma compreensão mais fácil e uma resposta mais rápida por parte do seu Spectrum, começa a adquirir alguma importância ser capaz de "falar" a sua própria linguagem. É fácil compreender a dificuldade de comunicar se, ao viajar no estrangeiro, se encontram apenas pessoas que falam a linguagem do local e você não a conhece. Sem recorrer a alguma linguagem à base de sinais o leitor não conseguiria fazer-se entender, e aliás essa linguagem de sinais seria afinal uma tentativa conjunta de encontrar uma linguagem comum que tanto o leitor como o seu interlocutor pudessem entender. Se fosse possível conseguir os serviços de um intérprete tudo se passaria mais facilmente. No entanto, tudo o que o leitor dissesse ao intérprete teria de ser interpretado por este e depois pelo interlocutor, pelo que a conversão seria sempre bastante limitada, e de qualquer modo muito lenta.

É também esta a situação que enfrentamos ao querer comunicar com o Spectrum — que contém um "intérprete" na sua memória. Tudo o que lhe "dizemos" em BASIC é interpretado para a sua linguagem própria, e vice-versa, existindo portanto um limite quanto ao que lhe podemos "dizer" e à velocidade a que esta comunicação se realiza. Quando os programas são escritos em BASIC, o Spectrum necessita de os guardar na memória e traduzi-los antes de os poder executar. Esta "interpretação" para código-máquina é necessária para permitir ao microprocessador Z80 funcionar — sendo o código-máquina a *única* linguagem que este é capaz de entender. Inevitavelmente, este processo de tradução de uma linguagem, obrigando a procurar o equivalente de todas as informações e dados, demora algum tempo. Se, por outro lado, tudo deve ser traduzido de BASIC para código-máquina antes de se executar qualquer instrução, o tempo perdido é bastante maior. Existem situações em que esta consideração pode ser importante, em particular, por exemplo, quando existem problemas de comando de caracteres gráficos ou de velocidade de execução de um programa.

Tal como quando se viaja no estrangeiro, se queremos de facto comunicar com rapidez, facilidade e rigor, só nos resta aprender a linguagem falada no país onde viajamos. O leitor resolveu viajar no "território" do seu computador, e portanto deve aprender a linguagem por ele falada. Pode parecer pouco razoável, mas pode acreditar em mim quando digo que o computador não vai certamente aprender português — pelo menos por enquanto!

Como se sabe, aprender uma língua estrangeira não é necessariamente muito fácil. Felizmente, a linguagem do computador é muito simples, apesar de o modo como a máquina a usa já não o ser; mas falaremos disto mais tarde. O computador pode somar um mais um, e como isto é a única coisa que sabe fazer passa a sua vida a tratar de zeros e uns. Para compreendermos melhor a razão de o computador apenas tratar zeros e uns, e como lhe é possível a partir de uma base tão simples realizar as tarefas mais complexas, é necessário compreender o modo como é constituído e consegue funcionar. Antes de o fazermos, verifiquemos no entanto a que velocidade consegue responder a instruções dadas em código-máquina em vez de BASIC.

Introduza no computador o programa seguinte:

```
5 BRIGHT 1: PAPER 4: BORDER 6
: INK 1: CLS
10 PRINT AT 5,8;"
15 PRINT AT 10,0;"
20 PRINT AT 13,0;"
```

```
 25 PRINT AT 0,0; "Guardar imagem"
 30 FOR x=0 TO 7167
 40 POKE (25600+x), PEEK (16384+x)
 50 NEXT x
 60 PRINT AT 0,0; "Transferência terminada"
 70 PAUSE 30
 80 CLS
 90 PRINT "Retorno da imagem"
100 FOR x=0 TO 7167
110 POKE (16384+x), PEEK (25600+x)
120 NEXT x
130 PRINT "FIM DO PROGRAMA"
```


O leitor convirá que este programa foi executado com bastante lentidão. Mas escreva agora a versão equivalente a este programa em código-máquina, e compare a velocidade a que ambas são executadas. Elimine as linhas 30, 40, 50, 70, 100, 110, 120 e acrescente as seguintes:

```
150 STOP
200 FOR 3=25501 TO 25524: READ a$
210 LET ans=(CODE a$(1)-48)*16
220 IF ans>9*16 THEN LET ans=ans-7*16
230 LET l=CODE a$(2)-48: IF l>9 THEN LET l=l-7
240 LET ans=ans+1: POKE a,ans
250 NEXT a
260 RETURN
300 DATA "21", "00", "40", "11", "00", "64", "01",
"00", "1C", "ED", "B0", "C9"
310 DATA "21", "00", "64", "11", "00", "40", "01",
"00", "1C", "ED", "B0", "C9"
```

```
 30 RANDOMIZE USR 25501

100 RANDOMIZE USR 25513

 80 PAUSE 100
```

Acrescente em seguida a linha:

```
1000 CLS : PAUSE 30: RANDOMIZE USR 25513: PAUSE
30: GO TO 1000
```

e faça executar o programa:

```
RUN 1000
```

Está convencido?

Voltemos ao modo como o computador funciona. Essencialmente, a máquina realiza cálculos aritméticos, usando um sistema de números binários e uma memória onde guarda informações. O seu Spectrum, enquanto computador digital, reconhece apenas DOIS estados; apenas pode reconhecer a presença ou au-

sência de um impulso eléctrico. Trabalha portanto apenas em termos de "ligado/desligado", e daqui a expressão "sistema binário". Tendo em conta este dado importante, podemos passar à fase seguinte que consiste em representar estes dois estados usando zeros e uns. É lógico representar a ausência de um sinal eléctrico por um "0", e a sua presença por um "1". Os dois estados do computador podem portanto ser representados do seguinte modo:

1. Existe um sinal? SIM — ou seja, "1"
2. Existe um sinal? NÃO — ou seja, "0"

Podemos comparar este modo de funcionar com o de um interruptor vulgar, que pode estar "ligado" ou "desligado". Quando está ligado, passa corrente eléctrica — quando está desligado não passa qualquer corrente. Pode parecer que reduzimos o computador de algo que é capaz pelo menos de contar até dois, a algo mais semelhante a um interruptor eléctrico. Até certo ponto isto é verdade, mas não contribui muito para vender o pobre computador. De facto, o seu Spectrum — ou mais exactamente o seu microprocessador Z80 — tem capacidade para ligar entre si uma série destas decisões ligado/desligado, produzindo assim um grande número de combinações diferentes tendo cada uma delas um significado particular. Para compreender melhor isto, considere por exemplo dois testes "interruptores" e combine-os. Poderá então obter quatro estados discretos:

1. Ligado e Ligado, que designaremos por 11
2. Ligado e desligado, que designaremos por 10
3. Desligado e Ligado, que designaremos por 01
4. Desligado e Desligado, que designaremos por 00

Combinando três destes interruptores, qualquer deles contendo apenas os estados Ligado/Desligado reconhecidos pelo computador, obtemos já oito estados diferentes:

1. Desligado e Desligado e Desligado 000
2. Desligado e Desligado e Ligado 001
3. Desligado e Ligado e Ligado. 011
4. Desligado e Ligado e Desligado 010
5. Ligado e Ligado e Ligado. 111

6. Ligado e Ligado e Desligado. 110
7. Ligado e Desligado e Desligado 100
8. Ligado e Desligado e Ligado. 101

O leitor deve ter já começado a aperceber-se da regra simples a que obedecem estas combinações. Se considerarmos dois estados específicos, estes podem ser combinados de um máximo de 2×2 (ou seja, 4) maneiras diferentes. Se partirmos de três tornam-se possíveis $2\times2\times2$ (ou seja, 8) combinações diferentes. Ora acontece que do mesmo modo que conseguimos ligar letras do alfabeto entre si, obtendo diferentes combinações que reconhecemos como "palavras" tendo cada uma o seu significado específico, também o computador pode ligar combinações de algarismos "0" e "1" produzindo "palavras" igualmente identificáveis. O seu Spectrum pode processar "palavras" constituídas por oito algarismos, e usando a regra das combinações vemos que isto permite obter um "vocabulário" de 256 palavras diferentes — $2\times2\times2\times2\times2\times2\times2\times2$, ou "dois elevado à oitava". Ligando os dois algarismos em causa deste modo, são criadas "palavras" que o computador é capaz de compreender — sendo cada "palavra" conhecida pelo nome de "número binário". O nome dado a este número binário é BYTE — e aos algarismos que o constituem chama-se BITS.

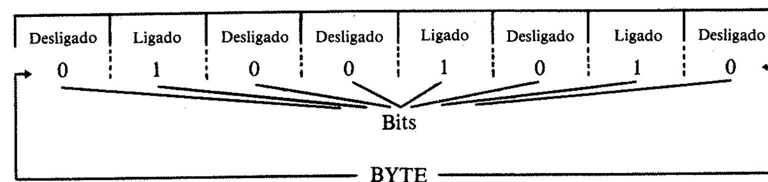


FIG. 1

Neste capítulo introdutório fizemos uma tentativa de ajudar o leitor a pelo menos começar a entender tanto a simplicidade como a sofisticação de um computador. O código-máquina é a linguagem que este usa, e portanto compreende, e referimos que esta pode ser reduzida a combinações de zeros e uns. O capítulo seguinte deter-se-á sobre o modo como os vários componentes são organizados no computador, tentando esclarecer o leitor sobre

aquilo que a máquina utiliza para funcionar, e como a chamada "aritmética binária" é usada de um modo tão eficaz. A tarefa seguinte consistirá em observar como são apresentadas as instruções antes de serem fornecidas ao computador de uma forma que este compreenda. Se bem que seja necessário também preocuparmo-nos com a forma como *nós* interpretamos estes números binários, não será necessário resignarmo-nos à terrível perspectiva de lermos sequências infinitas de zeros e uns, ou de termos de fornecer ao computador esta desagradável dieta...

SE AS BOAS MANEIRAS FAZEM O HOMEM, OS NÚMEROS FAZEM O COMPUTADOR...

II

PALAVRAS SOB A FORMA DE NÚMEROS

Na figura 2 vemos um diagrama simples representando o modo como o Spectrum se encontra organizado. Já fizemos uma referência ao microprocessador Z80, e a sua função consiste em *procurar, localizar* e *executar*. Este microprocessador, a CPU (Unidade de Processamento Central), como o nome indica, constitui o componente central do computador. Encontra-se em comunicação nos dois sentidos com os outros dois componentes: as entradas/saídas e a memória.

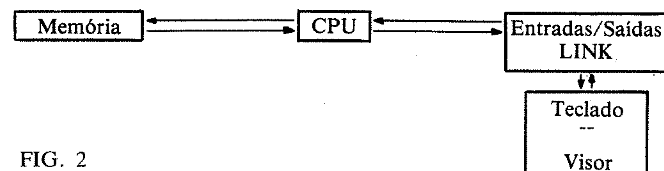


FIG. 2

As entradas/saídas: através de ligações físicas, a CPU pode dar a conhecer o resultado do seu labor a outros dispositivos — por exemplo um receptor de televisão ou um monitor vídeo — que apresentarão ao utilizador os resultados em causa. Alternativamente, permitem colocar estes resultados num suporte magnético onde são guardados, por exemplo a fita de uma cassette ou um suporte mais refinado como seja o disco. Estas ligações são ainda capazes de permitir a entrada (*input*) de informações de fora do computador — por exemplo através do teclado, ou vindas de uma memória externa. Torna-se então possível ligar o

microprocessador a circuitos que contenham sensores e comandem qualquer coisa desde a produção de uma linha de montagem até à interrupção do seu jogo de Invasores do Espaço para que o leitor saiba que deixou o seu jantar queimar-se ao lume! Chamaremos portanto a este sector INPUT/OUTPUT (entradas/saídas).

O outro componente é a memória central. Esta faz exactamente aquilo que seria de esperar — guarda informações e dados, que podem ser usados em qualquer momento. Porque recorda sem falhas tudo o que nela é guardado, é à memória que a CPU recorre quando necessita de descobrir o que deve fazer. Existem essencialmente dois tipos de memória — a RAM e a ROM.

RAM indica Memória de Acesso Aleatório, o que significa que quer queiramos "ler" ou "escrever" qualquer coisa no início ou no fim dessa memória necessitaremos exactamente do mesmo tempo. Isto é o contrário do que acontece noutros dispositivos de armazenamento de dados, como a cassette ou o disco, onde é necessário para ler um determinado registo passar por toda a informação que se encontra antes.

Por outro lado, usando uma fita em cassette, é possível transferir informações para a memória central.

Pelo contrário, ROM significa Memória Apenas de Leitura. Como seria de esperar, isto significa que só é possível ler o que está contido nela, e nunca escrever novos dados. Isto é semelhante ao que acontece no caso de um disco de música gravada, que nos permite ouvir a música mas não gravar novas composições. Note-se ainda que a ROM também é acedida de modo aleatório, tal como a RAM — a verdadeira diferença consiste em não ser possível introduzir informações novas na ROM.

Até agora fiz o possível por evitar falar muito em números, ao contrário do que é habitual quando se fala em computadores, porque tenho esperanças de conseguir manter o interesse do leitor... No entanto, os números têm um papel muito importante para estas máquinas e não é possível ignorá-los. Vejamos então o modo como os números são guardados e usados no computador, o que nos permitirá compreender o modo como podemos comunicar com o nosso Spectrum.

Voltemos assim à anterior analogia com os interruptores:

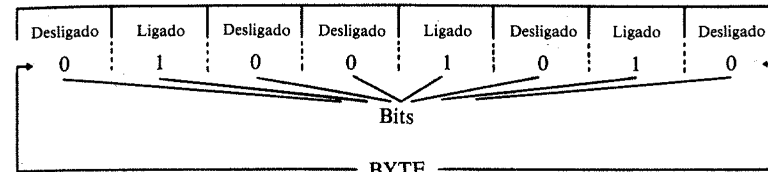


FIG. 3

É fácil compreender que o número binário 01001010 é uma "palavra binária" que se encontra dentro do vocabulário de 256 palavras aceites pelo Spectrum. Os extremos desta gama são representados por "tudo desligado" e "tudo ligado".

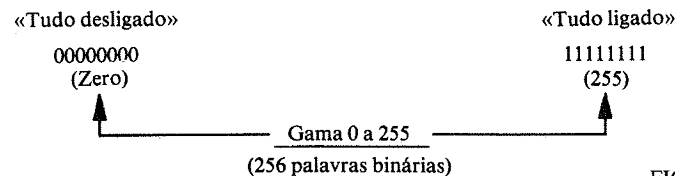


FIG. 4

Porque razão, no entanto, deve uma cadeia de oito zeros valer "zero" e uma de oito uns valer 255? A resposta a isto está no tipo de aritmética usada pelo computador — a *aritmética binária* — que talvez posssamos compreender melhor começando por explorar a estrutura do sistema "decimal" que nos é familiar.

O Sistema ou Base decimal utiliza dez algarismos (0 a 9). Emprega também uma notação "posicional", que permite calcular o valor de qualquer algarismo a partir da posição que este ocupa em relação à vírgula decimal. Procedemos do mesmo modo para saber o valor de um número em qualquer sistema — avaliamos cada um dos seus algarismos em relação a uma vírgula. Em aritmética binária, por exemplo, esta vírgula passa a ter o nome de vírgula "binária". Em todos os casos, esta vírgula separa a parte inteira de um número da sua parte fraccionária. Continuando por agora no sistema decimal, o valor de um algarismo num número é o produto dele próprio pela "potência de 10" determinada pela posição do número relativamente à vírgula decimal. Os algarismos do número estão dispostos em "colunas", cada uma delas tendo um valor dez vezes superior ao da coluna imediatamente à direita.

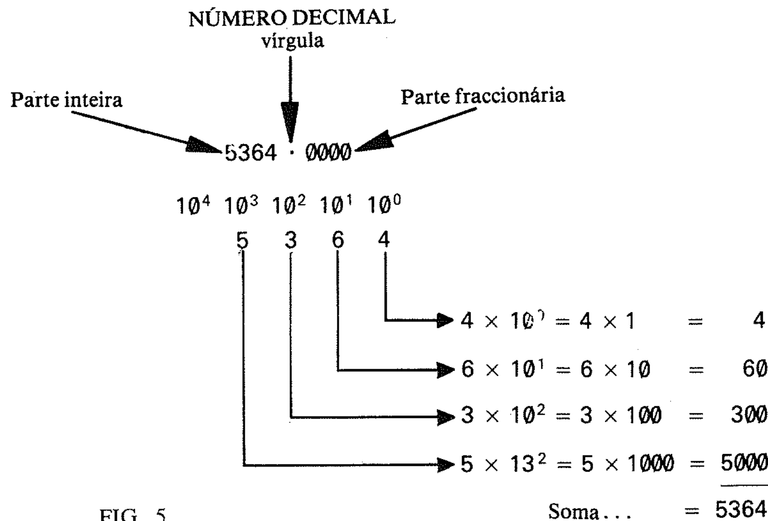


FIG. 5

Cada algarismo/coluna possui uma "potência" dez vezes superior à do algarismo/coluna imediatamente à direita.

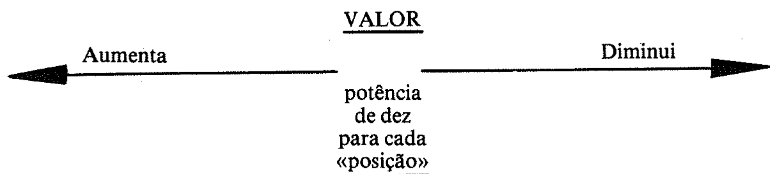


FIG. 6

O sistema binário baseia-se apenas no uso de dois algarismos (0 e 1), tendo portanto uma *base dois*. Tal como no sistema decimal emprega-se um sistema de notação posicional, mas desta vez cada algarismo possui um valor equivalente a uma "potência de dois", duas vezes superior à do algarismo imediatamente à sua direita:

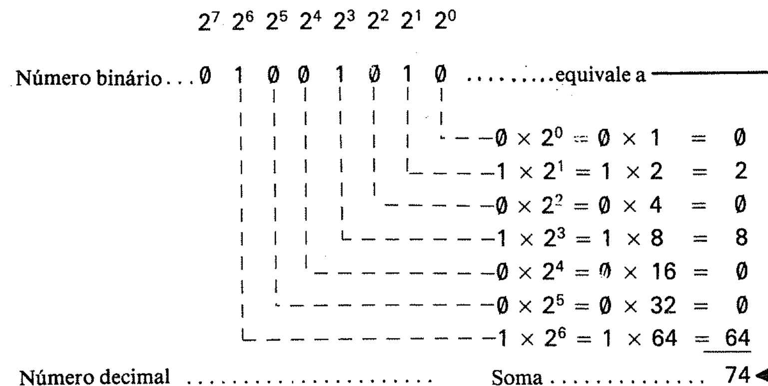


FIG. 6ᴬ

Compreendemos agora que no exemplo apresentado na figura anterior 01001010 é de facto equivalente ao decimal 74. Do mesmo modo, 00000000 = Zero (decimal) e 11111111 = 255 (decimal).

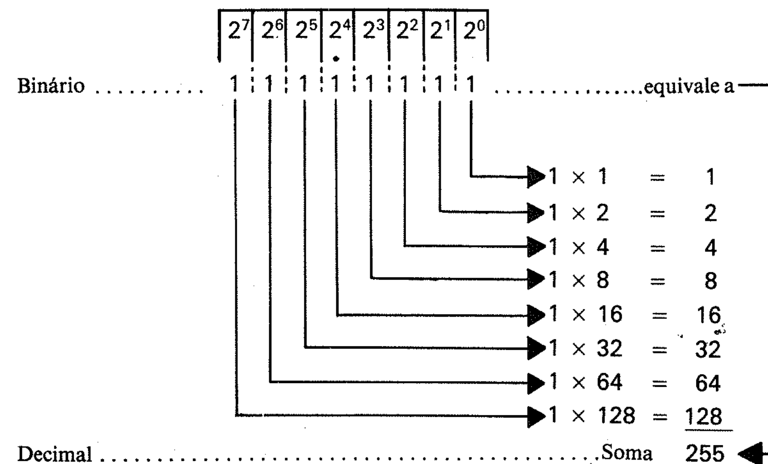


FIG. 7

Se bem que o computador utilize o equivalente binário do número decimal que deseja utilizar, prometi ao leitor no último capítulo que não lhe seria necessário recordar cadeias de zeros e uns para poder comunicar com o Spectrum. Infelizmente, porém, o leitor também não poderá confiar nos familiares números decimais para este efeito. No entanto, enquanto que a "base 10" é pouco útil e a "base 2" é impraticável, a "base 16" é extremamente conveniente. Não desespere — tudo isto é de facto mais fácil do que parece. Aplicando os princípios já referidos, a base 16 — ou *hexadecimal* — emprega nada menos de 16 algarismos. Os primeiros dez são iguais aos do sistema decimal (0 a 9), e os seis restantes são representados pelas primeiras letras do alfabeto — A, B, C, D, E, F. A tabela que se segue define a relação entre números binários, decimais e hexadecimais até 15 (decimal).

Número binário	*Equivalente decimal*	*Equivalente hexadecimal*
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

No apêndice A encontra-se uma tabela dos equivalentes binário/decimal/hexadecimal até 255.

Consideremos novamente o exemplo de 74 (decimal) e do seu equivalente 01001010 (binário), e notemos o modo prático como pode ser convertido em hexadecimal (muitas vezes referido apenas por "hex"):

Verifiquemos o resultado:

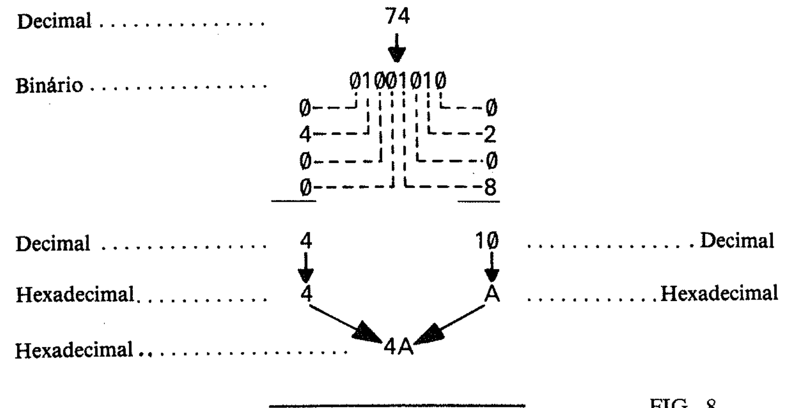


FIG. 8

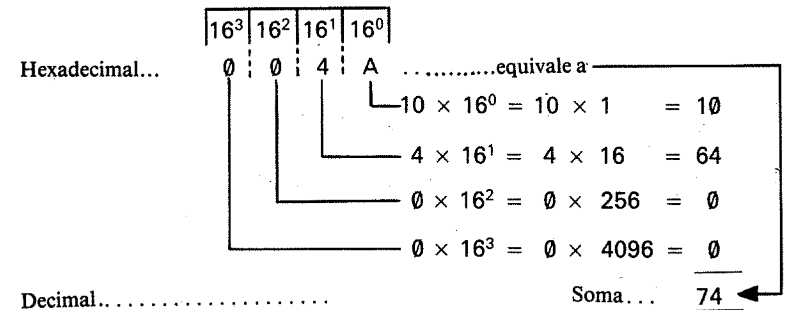


FIG. 9

Número binário	*Equivalente decimal*	*Equivalente hexadecimal*
01001010	74	4A

Se procedermos do mesmo modo para 255, o maior número que pode ser tratado pela CPU do Spectrum, notamos uma vez mais a facilidade de representação deste número no sistema hexadecimal:

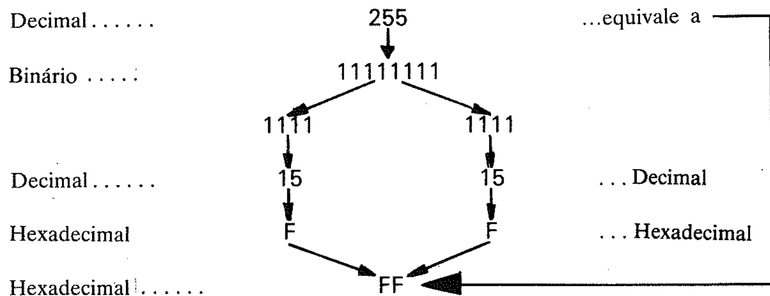


FIG. 10

Número binário	*Equivalente decimal*	*Equivalente hexadecimal*
11111111	255	FF

A divisão do número binário em grupos de 4, e a combinação dos equivalentes decimais de cada um destes, torna-se extremamente prática como método de representação dos "endereços" das *posições de memória*. Até agora sublinhámos bastante o facto de 255 ser o maior número que a CPU pode tratar. No entanto existem modos de ultrapassar isto; de facto, a CPU consegue utilizar a sua capacidade para tratar palavras binárias *juntamente* com a sua capacidade para somar um e um, e nestas condições consegue processar números até 65535! O tratamento de números entre 0 e 65535 requer o uso de números binários com 16 algarismos (bits, ou algarismos binários), sendo o maior destes 1111 1111 1111 1111 — o equivalente binário de 65535 (verifique isto se tiver dúvidas). Todos os números envolvidos aqui, quer sejam em binário ou em decimal, tendem a tornar-se demasiado grandes. O valor particular do sistema hexadecimal torna-se agora evidente — em particular porque se podem considerar os algarismos binários em grupos de 4, representá-los em seguida por um número hexadecimal de um único algarismo, e combinar depois estes algarismos obtendo um número de aparência muito mais simples:

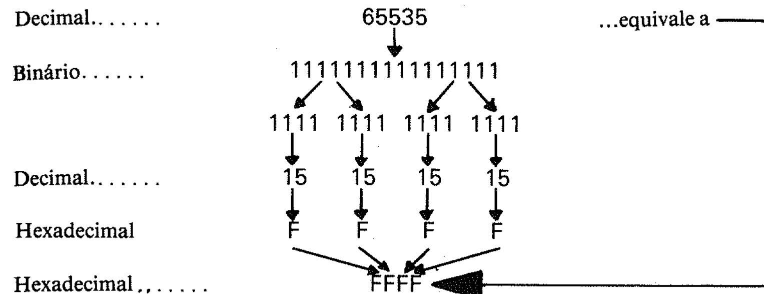


FIG. 11

Número binário	*Equivalente decimal*	*Equivalente hexadecimal*
1111111111111111	65535	FFFF

Qualquer número na gama 0 a 65535 pode agora ser reduzido a um número hexadecimal de quatro algarismos, na gama 0000 a FFFF.

Tal como acontece com todos os "sistemas de numeração", podem-se realizar cálculos aritméticos em aritmética binária e, como é óbvio, o computador serve precisamente para fazer isso. Tudo o que faz reduz-se a somar ou subtrair — o Spectrum consegue até fazer as subtracções somando... mas voltaremos a este assunto mais adiante. A vantagem disto é que não temos de nos preocupar excessivamente com a multiplicação e a divisão. Em termos gerais, as convenções (ou regras) que se aplicam em aritmética decimal aplicam-se igualmente bem em aritmética binária. Assim, o maior número que pode ocorrer em qualquer coluna é 1; isto é, um menos do que o número total de algarismos permitidos no sistema (dois menos um). Em decimal, o maior algarismo é 10-1 = 9. Quando se realizam somas entre dois números binários, a familiar regra do "e vai um" (transporte de uma unidade para a coluna seguinte) aplica-se quando a soma de dois algarismos é superior a 1. A consulta da tabela que se segue clarifica um pouco estas "verdades" básicas:

0+0= 0
0+1= 1
1+0= 1
1+1= 10 ("zero, e vai um")

+	0	1
0	0	1
1	1	10

Como exemplo, a soma de dois números binários teria a seguinte aparência:

	01001010	(Decimal 74)
	+00111010	(Decimal 58)+
Transporte.........	0111010–	
Soma.............	10000100	(Decimal 132)

Aplicando os mesmos princípios à subtracção (exceptuando o facto de neste caso tirarmos dois, em vez de acrescentarmos um), verificamos que esta pode ser considerada como uma inversão exacta do processo de soma:

	10000100	(Decimal 132)
	–00111010	(Decimal 58)–
Tirar	01111010–	
	01001010	(Decimal 74)

Este processo é de facto um pouco mais complicado do que é necessário, sendo mais prático executar as subtracções usando um método conhecido pelo nome de "complemento para dois" — sendo isto que o computador faz na realidade. No exemplo anterior, o resultado da subtracção pode ser obtido mais facilmente *somando* o "complemento para dois" de 00111010 (o diminuidor) a 10000100 (o diminuendo). Para determinar o "complemento para dois" de qualquer número binário substituem-se muito simplesmente os zeros por uns, e os uns por zeros, somando depois um ao resultado:

Diminuendo ...... 00111010 (Decimal 58)
Complemento para 2 ...... 11000101 + 00000001
= 11000110

Diminuidor .... 10000100 (Decimal 132)
+
Complemento para 2 11000110
Soma 101001010

«Bit de sinal»

Decimal 74

FIG. 12

Os oito bits menos significativos são os mesmos de anteriormente (decimal 74), se bem que exista agora um "algarismos de transporte" adicional — conhecido pelo nome de Bit de Sinal. Este pode ser menosprezado em termos de determinar o resultado numérico, mas é tido em conta como indicador do sinal positivo ou negativo do resultado do cálculo. Quando o Bit de Transporte ou Sinal é 1, isto significa que o número resultante tem um valor positivo. Pelo contrário, se o Bit de Transporte é 0, o resultado do cálculo terá um valor negativo.

OS COMPUTADORES SÃO UMA ESPÉCIE DE ARRANHA-CÉUS

III

OS ENDEREÇOS, E COMO LÁ CHEGAR...

Vale a pena resumir o que dissemos até agora. Começámos pela atitude bastante genérica de descrição do código-máquina como linguagem própria do computador, e considerámos os benefícios que decorrem da aprendizagem dessa linguagem. Do mesmo modo, partindo de uma ideia muito geral do modo como o Spectrum funciona, começaremos a estudar em algum pormenor o modo como os seus vários componentes se ligam entre si e utilizam a aritmética binária para funcionarem. Isto obrigou-nos a conhecer um outro sistema de números — hexadecimal — para podermos aprender a reduzir os números binários de oito algarismos — bits — a números hexadecimais de dois algarismos, ou números binários de 16 bits a números hexadecimais de quatro algarismos.

Tendo referido o limite do Spectrum em termos de usar apenas um vocabulário de 256 palavras, indiquei no entanto que é capaz de guardar informações em mais de 65000 posições diferentes. Este sistema de "endereçamento" será melhor estudado durante este capítulo. Além disso veremos em maior detalhe o modo como a memória central do Spectrum está estruturada e organizada, e começaremos a aprender o modo de manipular o conteúdo dessa memória.

A nossa primeira tarefa será explorarmos a estrutura e organização da memória do Spectrum consultando o seu Mapa de Memória (figura (13).

Cada posição de memória é um local onde é possível guardar informações, sendo vulgarmente designada por um *endereço*. Esta terminologia não foi desenvolvida por acaso, mas sim em correspondência a um conceito que nos é familiar. A cada posição é atribuído um número na gama 0 a 65535 — transforman-

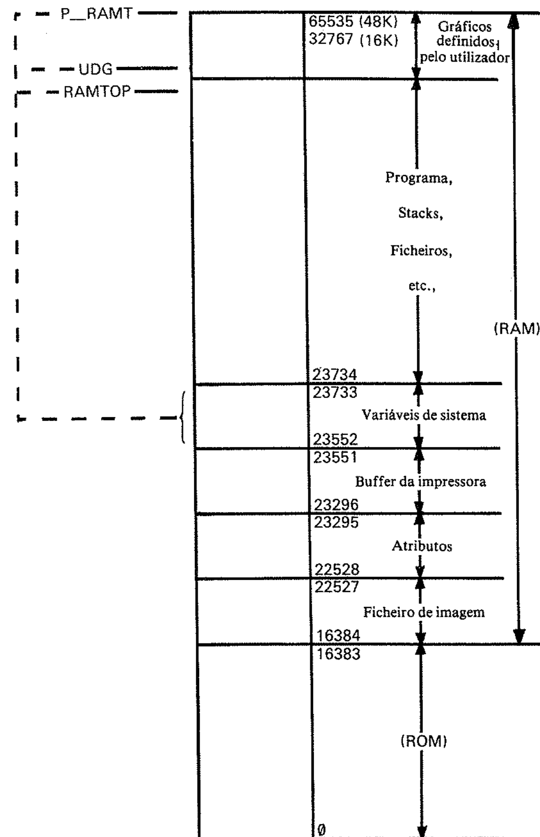


FIG. 13

do-se assim este número no endereço específico dessa posição de memória. Observando o Mapa de Memória, verificamos que os primeiros 16384 (ou 16 K) endereços existem em ROM. Como já vimos estas iniciais indicam tratar-se de uma memória apenas de leitura, formada essencialmente por um bloco de 16384 posições onde a informação foi armazenada no momento em que a própria ROM foi fabricada. Não podemos portanto alterar o seu conteúdo. Na versão do Spectrum com 16 K existem ainda outras 16384 posições em RAM. A versão de 48 K utiliza o número máximo de posições que podem ser endereçadas pela CPU Z80;

além dos 16 K em ROM existem então 48 K de RAM (posições 32767 a 65535, ou exprimindo em hexadecimal 8000 a FFFF). No conjunto somam um total de 64 K — ou 65536 bytes.

Antes de prosseguir gostaria de eliminar qualquer confusão que subsista relativamente ao número "65535". O leitor já terá compreendido que a CPU do Spectrum pode processar 256 bytes diferentes de 8 bits cada. Considere agora, hipoteticamente, que a memória da máquina é uma espécie de enorme arranha-céus de apartamentos — com 256 andares, e tendo cada andar 256 apartamentos diferentes. Cada um destes poderia dispor de um endereço baseado no andar onde se situa e na posição que ocupa neste. Considerando que os andares são numerados sequencialmente de 0 a 255, e os apartamentos também de 0 a 255 em cada um dos 256 andares referidos, torna-se possível localizar qualquer apartamento específico indicando os números de andar e de "porta" que lhe correspondem. Por exemplo, andar 10, porta 54. A posição desse apartamento relativamente a todos os outros poderia ser então determinada do seguinte modo:

$$(\text{Número do andar} \times 256) + \text{Número da porta} = (10 \times 256) + 54 = 2614$$

Do mesmo modo, o último apartamento seria $(256 \times 256) + 255 = 65535$. É de facto deste modo que o Spectrum endereça as suas 65536 posições de memória — se bem que, o que é compreensível num produto Sinclair baseado no Z80 — o faça "de pernas para o ar". Utiliza dois bytes para guardar o endereço; no primeiro guarda a parte "menos significativa" deste (54 no exemplo anterior) e no segundo guarda a sua parte "mais significativa" (10 no exemplo). Os códigos de endereço de 16 bits usados pela CPU Z80 são de facto formados por dois bytes contendo um número bastante comprido; os primeiros oito bits constituem o byte menos significativo, e o segundo o byte mais significativo — a que chamaremos abreviadamente, usando expressões inglesas, LSB e MSB.

Isto pode ser exemplificado (e utilizado) através de uma fórmula bastante simples que determina quantos bytes de memória foram usados para guardar um programa Basic. Carregue um programa na máquina, e em seguida determine os bytes ocupados do seguinte modo:


```
PRINT (PEEK 23653 + (PEEK 23654) * 256) — (PEEK
23635 + (PEEK 23636) * 256).
```

Esta fórmula determina simplesmente o ponto onde se inicia a memória livre (indicada pela variável de sistema STKEND) e subtrai-lhe o endereço inicial do programa Basic. Estes endereços são determinados lendo — através da função PEEK) o conteúdo das variáveis de sistema apropriadas — que guardam endereços em dois bytes, começando pelo LSB (menos significativo). O primeiro endereço é portanto determinado observando a posição de memória 23653, extraindo o seu conteúdo (um byte de oito bits) e somando-lhe o conteúdo da posição de memória seguinte multiplicado por 256 (visto que se trata do byte mais significativo). O segundo endereço é determinado de forma semelhante.

Ao explorar estes endereços utilizámos a função PEEK (espreitar, inspeccionar), que em inglês descreve exactamente o que fizemos. Permite-nos observar uma posição de memória e descobrir o que nela se encontra — um pouco como se olhássemos pela janela de um dos 65536 apartamentos do nosso arranha-céus. Na parte da memória em ROM só podemos fazer isto — olhar mas não tocar... Na parte de RAM, podemos no entanto olhar e tocar. A ordem para "tocar", ou mais exactamente alterar o conteúdo de uma posição de memória em RAM é POKE; esta função permite ao utilizador modificar o conteúdo de uma posição de memória introduzindo nela um novo valor. Peça ao seu Spectrum que imprima PEEK n, onde "n" seja uma posição de sua escolha (em RAM). Depois de ter descoberto o conteúdo dessa posição (de ter sido impresso no visor) faça POKE n,x — onde "n" é ainda essa posição e "x" é um valor também escolhido por si, e na gama 0 a 255. Em seguida faça novamente PEEK n e confirme que alterou de facto o conteúdo desse endereço. Até ao fim deste livro vai dar bastante uso a estas duas funções...

Talvez sem o ter compreendido ainda, o leitor chegou a um ponto em que está prestes a começar a falar com o Spectrum na sua própria linguagem. Sabemos agora de que são constituídas as "palavras" dessa linguagem, onde estão guardadas, e como podemos introduzi-las no computador — usamos a função POKE. Que outras questões básicas necessitamos ainda de conhe-

cer? Temos evidentemente de preparar o computador para receber as nossas instruções, e como é óbvio será necessário determinar exactamente quais as palavras que nos interessa fazer compreender e executar pelo computador. Num sentido estrito, o microprocessador Z80 deveria ser capaz de compreender apenas 256 instruções de código-máquina, mas de facto, o que já não nos surpreende, consegue "estender" as suas aptidões e, através de um uso judicioso de "prefixos", reconhece de facto mais de 600. Todas estas instruções se encontram listadas no Apêndice B. Passemos agora a alguns princípios básicos.

As rotinas em código-máquina podem sempre ser chamadas através de um programa Basic que utilize a ordem "RANDOMIZE USR". Assim, para ganhar acesso a uma rotina em código-máquina emprega-se a subrotina USR, especificando o ponto da memória onde aquela se encontra. Por exemplo, RANDOMIZE USR 30000 chamará uma rotina em código-máquina guardada a partir da posição de memória 30000 — estando o resto da rotina nos bytes que se seguem a este. Como se determina então o byte inicial? Bom, somos nós próprios que introduzimos o princípio da rotina num determinado byte e em seguida informamos disso o computador. Fazendo CLEAR 29999 (ou qualquer outro valor conveniente) passamos a RAMTOP para essa posição, o que significa que só dispomos da memória disponível até 29999 para introdução e execução de um programa Basic. A partir daí podemos introduzir um programa em código-máquina — começando pela posição de memória seguinte, 30000. A ordem NEW, usada para limpar a memória da máquina, apenas o faz até à RAMTOP que tiver sido definida, pelo que o código-máquina que se encontra acima está protegido contra o uso desta ordem e não pode ser corrompido por um programa Basic que se encontre na máquina, pois este nunca utilizará as posições de memória acima da RAMTOP. A ordem RUN limpará as variáveis do programa Basic, mas também sem alterar a RAMTOP. Nestas condições, usar CLEAR com um endereço à frente constitui um modo de mover a RAMTOP "para cima" obtendo mais espaço para um programa Basic (incluindo por exemplo a área dos gráficos definidos pelo utilizador, nos últimos bytes da memória) ou "para baixo" de modo a aumentar a quantidade de memória RAM protegida contra esse mesmo programa.

Tendo decidido onde queremos guardar o nosso programa em código-máquina, como poderemos fazê-lo? Como se disse anteriormente, este código deve ser guardado sequencialmente porque é por esta mesma ordem que a CPU do Spectrum o irá executar. O processo de procura e execução das instruções em código começa pelo endereço indicado à máquina, passando depois às sucessivas posições de memória — interpretando tudo o que encontra pelo caminho. Este processo continua até a máquina receber uma instrução para parar — ou até ter um esgotamento nervoso... Em seguida apresentamos um pequeno programa que demonstra este modo "sequencial" de trabalhar. Não se preocupe com o modo como trabalha — fundamenta-se completamente no facto da ROM do Spectrum ter entrado em transe!


```
 5 RESTORE
10 READ n
20 LET a =120
25 PLOT 55,27: DRAW a,a,n*PI
32 CLS
33 IF n=9999 THEN GO TO 5
35 GO TO 10
40 STOP
50 DATA 597,631,315,343,297,631,613,787,313,741,187,
147,279,9999
```

Terminada esta diversão, observemos agora quatro opções quanto ao modo de carregar código-máquina no computador:

1. POKE um byte de cada vez.
   POKE X,Y

Onde X é o endereço inicial e Y é o valor a entrar. O segundo byte é introduzido na posição X+1, o terceiro na X+2, etc.

2. Ciclo FOR/NEXT
   10 FOR A=1 TO X
   20 INPUT Y
   30 POKE A + posição inicial, Y
   40 NEXT A

3. READ (Ler) Dados
   10 FOR A=1 TO X

```
20 READ N
30 POKE posição inicial + A,N
40 NEXT A
50 DATA N1,N2,N3,N4,...
```

onde N1, N2, N3, N4,... são os bytes da rotina em código-máquina, escritos sequencialmente.

4. LOAD (Carregar) da cassette.
   CLEAR posição de memória
   LOAD "nome" CODE

As opções 2 e 3 oferecem duas bases diferentes para escrita de um programa carregador de código-máquina. Como é óbvio, carregar a máquina em códigos binários é definitivamente de evitar, e os nossos conhecimentos recentes de hexadecimal podem ser aqui usados para facilitar o trabalho. Vejamos um carregador hexadecimal:

```
  10 PRINT "***Carregador de código-máquina***"
  20 PRINT "         © James Walsh 1963"
  30 INPUT ' "Endereço inicial?";endereço
  40 PRINT ' ' "Indique código um por um"
  50 PRINT "sempre em maiúsculas"
  60 PRINT endereço; "=";
  70 INPUT a$
  80 IF a$="END" THEN GO TO 1000
  85 IF LEN a$<>2 THEN GO TO 70
  90 LET hi=CODE a$(1)-48
 100 IF hi>9 THEN LET hi=hi-7
 110 LET ans=16*hi
 120 LET low=CODE a$(2)-48
 130 IF low>9 THEN LET low=low-7
 140 LET ans=ans+low
 150 PRINT a$
 160 POKE endereço, ans
 170 LET endereço=endereço+1
 180 GO TO 60
1000 PRINT AT 21,0; "Programa terminado"
```


ALGUMAS MÁQUINAS REGISTADORAS NÃO SERVEM PARA GUARDAR DINHEIRO...

IV

TRABALHAR COM OS REGISTOS

Tendo lido os capítulos anteriores, o leitor deve já ter uma ideia razoável sobre a gama de números que o computador pode compreender, o modo como se pode usar esses números, e como estes são usados pela máquina permitindo-nos manipular a sua memória; tudo se passa como se o computador fosse constituído por uma longa fiada de caixas vazias — todas elas com números específicos, permitindo-nos acedê-las sem termos de passar por todas elas — nas quais se pode colocar ou tirar "coisas". O que faremos neste capítulo é precisamente descobrir o modo como o computador manipula números quando se encontram fora das caixas, e o modo como os podemos utilizar. Observaremos o modo como o computador guarda os números, como guarda as instruções dadas em código-máquina, e estudaremos algumas das instruções mais simples existentes em código-máquina para este fim. Em seguida começaremos a explorar o modo de utilizar estas instruções nos nossos programas. Apresentaremos ainda ao leitor um problema que este deverá resolver usando um programa em código-máquina.

Não há mal nenhum em pôr e tirar números de uma caixa, mas de facto não haverá grande utilidade em fazê-lo se tudo ficar por aí... Quando escrevemos um programa Basic utilizamos uma variável. Por exemplo, se quisermos guardar um número identificado pela letra A escrevemos simplesmente LET A=número. Depois podemos fazer o que quisermos a essa variável A. Podemos passá-la a zero escrevendo simplesmente LET A=0.

Infelizmente não existem variáveis em código-máquina. Em vez disso temos aquilo a que se chama REGISTOS. Existem sete

registos de um único byte, fáceis de usar e manipular. Existem ainda outros, mas são principalmente usados pelo computador. Podemos sem dúvida usá-los, mas com maior dificuldade e é por essa razão que os deixaremos para mais tarde. Durante grande parte do tempo que gastará a programar em código-máquina, o leitor usará apenas estes sete registos "fáceis". Cada um deles é identificado por uma letra: A, B, C, D, E, H, L. Não há nenhuma razão para se usarem estas letras em vez de outras, pelo que não vale a pena perder tempo a tentar descobrir alguma lógica que as justifique. Como o leitor já sabe, o computador, quando manipula um único byte, só pode tratar números entre 0 e 255. Como qualquer destes registos é constituído por um só byte, 255 é o número máximo que neles pode ser guardado. O leitor já notou certamente que este número é demasiado pequeno para ter qualquer valor prático — quanta vezes consegue o leitor escrever um programa sem qualquer número superior a 255? Para remediar isto, o computador agrupa dois registos obtendo aquilo a que se chama um par de registos. Se se combinam dois registos deste modo, o número máximo que ambos podem conter é 256×256, menos 1, o que dá 65535. Em terminologia de computadores esta gama 0 a 65535 é representada por 64 K. A razão que nos obriga a diminuir "1", é que de facto os números variam entre 0 e 65535, isto é, que as 65536 combinações possíveis devem englobar também o número zero.

Os registos não podem ser combinados entre si de uma forma qualquer. Existem modos bem definidos de os combinar: BC, DE, HL, como se pode ver na figura 14.

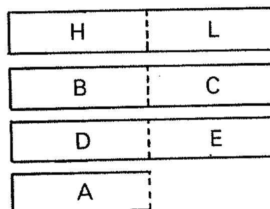


FIG. 14

Se o leitor observar rapidamente a lista de ordens em código-máquina e os respectivos códigos no fim do livro, notará sob o título "Mnemónicas" a existência de um vasto número de ordens que utilizam directamente estes registos. É então óbvio que a compreensão destes registos e das suas funções lhe será mais útil do que tudo o resto para a programação em código. Uma potencialidade da CPU Z80 que se mostrará muito valiosa, é a de usar os registos como registos simples, ou seja, de um só byte numa gama 0 a 255, ou juntos dois a dois, numa gama 0 a 65535. É nisto que se fundamenta a versatilidade do computador.

Em resumo, vamos ver como é possível carregar números nestes registos, e como podemos somá-los ou subtraí-los. Mais tarde estudaremos igualmente o modo de realizar manipulações mais complicadas nestes registos. Enquanto outros registos nos são pouco úteis, e alguns são reservados até ao uso do próprio computador, haverá momentos em que o acesso a estes registos a partir dos nossos programas será particularmente útil.

Provavelmente o melhor modo de entender o conceito de "registos" consiste em pensar no computador como em alguém que usa um casaco. Este possui sete algibeiras, cada uma delas com uma etiqueta. O computador pode facilmente tirar coisas destas algibeiras, ou pôr coisas lá dentro. Para o fazer é necessário dar-lhe uma ordem porque não tem capacidade para pensar sozinho, o que nos obriga a fornecer à máquina instruções rigorosas. O computador só consegue compreender ordens que lhe são dadas sob a forma de números, ou códigos. A linguagem cujas instruções são integralmente representadas por números é conhecida pelo nome de linguagem-máquina. Infelizmente, se bem que seja razoavelmente fácil introduzir ordens na memória de um computador sob a forma de números — usando a função POKE — tentar recordar o código correspondente a uma dada ordem não é muito simples para nós, mortais. Não desespere, porém, porque como este problema já existe há muito arranjou-se uma solução. A resposta para o nosso problema surge sob a forma da "linguagem *assembly*". Esta linguagem consiste num conjunto de instruções exactamente correspondentes aos códigos da linguagem-máquina, mas escritas de um modo mais fácil de compreender por nós. Por exemplo, uma instrução que em código-máquina tem a aparência 01101011 terá como equivalente em linguagem assembly LD A,1 (ou seja, "load" — carregar — no registo A o número 1).

Se bem que estas expressões possam não ter grande significado para o leitor neste ponto, é óbvio que a nossa maior facilidade

em usar grupos de letras e números, à qual recorremos sistemati-
camente, nos permite usar mais facilmente esta linguagem as-
sembly do que cadeias de zeros e uns. Isto torna-se muito mais
óbvio quando tentamos colocar na memória (na nossa, entenda-
-se, não na do computador) grupos de números. As representa-
ções "humanizadas" das ordens binárias utilizadas em lingua-
gem assembly são designadas por *mnemónicas* (se bem que infe-
lizmente só o sejam para quem entender inglês...)

Não podemos escrever as mnemónicas directamente no tecla-
do do computador — muito simplesmente porque a máquina
não percebe as nossas mnemónicas — e portanto temos de
convertê-las para a forma que o pobre computador entende. Isto
significa reconverter novamente para códigos binários. Existem
duas formas básicas de o fazer. Uma, perfeitamente eficaz mas
bastante demorada, consiste em recorrer às tabelas de conversão
apresentadas no final do livro. Um método mais fácil, e portan-
to mais rápido, consiste em escrever um programa ASSEM-
BLER.

À medida que avançarmos neste livro serão evidentemente in-
troduzidas as instruções relevantes, juntamente com as corres-
pondentes mnemónicas. Explicarei o significado e o uso de cada
uma delas, e ao utilizar os exemplos o leitor estará a realizar con-
versões de "mnemónicas" para "código" e carregando estes no
computador usando o programa "carregador" fornecido. Pode-
rá sem dúvida verificar que um programa Assembler, capaz de
traduzir as mnemónicas em código sem nos forçar a perder tem-
po com o assunto, é extremamente útil, mas não necessita de sair
já porta fora para comprar um. Se decidir eliminar parte do tra-
balho, ou dispor de um Assembler quando eu falar destes pro-
gramas mais adiante, pode verificar numa loja quais os existen-
tes e escolher um deles.

É importante recordar que a linguagem assembly não é uma
adaptação do código-máquina — ao contrário da Basic. Em lin-
guagem assembly existe apenas uma mnemónica para cada ins-
trução em linguagem-máquina, e vice-versa — só existe uma ins-
trução em linguagem-máquina para cada mnemónica da lingua-
gem assembly. Podemos portanto afirmar que a linguagem as-
sembly é de facto equivalente à linguagem-máquina. Quase
todas as mnemónicas indicam abreviadamente as operações
comandadas pelas correspondentes instruções em código. É por-

tanto fácil converter estas abreviaturas em instruções. Por exem-
plo: INC HL significa "incrementar" HL. Do mesmo modo,
LD A,0 significa "load" (carregar) em A o número 0. Os códi-
gos-máquina equivalentes a estas duas instruções são 23 e 62,0
respectivamente.

Cada pessoa lista os seus programas de um modo diferente,
em linguagem-máquina ou em assembly. É muito mais fácil
compreender um programa que se encontra escrito em lingua-
gem assembly, mas um Assembler para o Spectrum tende a ser
relativamente caro. Se resolver não adquirir um Assembler, ou
não dispuser de um por qualquer razão, continua a ser fácil car-
regar um programa na máquina utilizando um "carregador he-
xadecimal". Para tornar isto possível, resolvi listar todos os pro-
gramas deste livro nas duas formas, de tal modo que seja qual
for o modo como quiser introduzir o programa na máquina con-
tinua a poder consultar a versão em linguagem assembly para
descobrir o que está a fazer.

Não se esqueça de que, como o código-máquina se baseia na
própria CPU, que neste caso é o microprocessador Z80, se sou-
ber escrever código-maquina para o Spectrum não necessitará de
muito esforço para o escrever também para o ZX81, o ZX80 ou
o Lynx — ou aliás para qualquer microcomputador que utilize a
CPU Z80. É particularmente importante ter isto em conta quan-
do se progride relativamente ao Spectrum — ou se decidir au-
mentar a sua colecção de micros! Se por exemplo resolver com-
prar um BBC ou um Commodore Vic 20, será obrigado a apren-
der novamente código-máquina para usar nesses computadores.
Esta não é a única consideração a ter em conta ao decidir com-
prar um novo computador, mas pode ser um factor importante.

Bom, façamos agora um intervalo. Vamos estudar um interes-
sante programa que actua como um exemplo simples daquilo
que o código-máquina realmente é. Note que existem duas partes
neste programa — a listagem em linguagem assembly e a lista-
gem em linguagem-máquina. O programa serve para deslocar os
pixels para a esquerda (*scroll* lateral).

```
org 30000
30000 21 FF 57        ld hl,22527
30003 E5              push hl
30004 11 20 5B        ld de,23328
30007 01 20 00        ld bc,32
```

```
30010 ED B8             lddr
30012 E1                pop hl
HL = Ender. inf. do bloco inferior
B = Número de Blocos
30013 06 03             ld b,3
A
30015 C5                push bc
30016 06 08             ld b,8
B
30018 C5                push bc
30019 E5                push hl
30020 06 08             ld b,8
C
30022 C5                push bc
30023 E5                push hl

30024 E5                push hl
30025 D1                pop de
30026 25                dec h
30027 01 20 00          ld bc,32
30030 ED B8             lddr
30032 E1                pop hl
30033 25                dec h
30034 C1                pop bc
30035 10 F1             djnz,C
30037 24                inc h
30038 11 E0 06          ld de,1760
30041 E5                push hl
30042 19                add hl,de
30043 D1                pop de
30044 01 20 00          ld bc,32
30047 ED B8             lddr
30049 E1                pop hl
30050 11 20 00          ld de,32
30053 ED 52             sbc hl,de
30055 C1                pop bc

30056 10 D8             djnz,B
30058 11 00 07          ld de,1792
30061 ED 52             sbc hl,de
30063 E5                push hl
30064 11 20 00          ld de,32
30067 19                add hl,de
30068 E5                push hl
30069 D1                pop de
30070 E1                pop hl
30071 E5                push hl
30072 01 20 00          ld bc,32
30075 ED B8             lddr
30077 E1                pop hl
30078 C1                pop bc

30079 10 BE             djnz,A
30081 11 20 00          ld de,32
30084 19                add hl,de
30085 EB                ex de,hl
30086 21 20 5B          ld hl,23328
30089 06 20             ld b,32

D
30091 7E                ld a, (hl)
30092 00                nop
ld a,0 para não reentrada da imagem
30093 12                ld (de),a
30094 1B                dec de
30095 2B                dec hl
30096 10 F9             djnz,D
30098 C9                ret
```


Instruções simples sobre registos

O leitor já está familiarizado com os modos como os registos simples se podem combinar em pares de registos. É provável que tenha chegado o momento de recordar-lhe que os registos simples podem tratar números entre 0 e 255, e que os pares de registos podem tratar números entre 0 e 65535. Comecemos por ver como se carregam números nos registos simples.

Consideremos por exemplo que se deseja colocar um número — por exemplo 11 — num dos registos (ou "algibeiras"). Para o fazer temos de carregar este número na "algibeira", o que é fei-

to recorrendo a uma instrução simples conhecida por LD, abreviatura de "Load" (carregar). Dado que existem várias algibeiras, é necessário que exista um número igual de códigos, cada um deles carregando números numa delas. Os códigos correspondentes às mnemónicas "load" são indicados em seguida:

Código de operação	Hexadecimal	Decimal
LD A,xx	3Exx	62,xx
LD B,xx	06xx	6,xx
LD C,xx	0Exx	14,xx
LD D,xx	16xx	22,xx
LD E,xx	1Exx	30,xx
LD H,xx	26xx	38,xx
LD L,xx	2Exx	46,xx

A primeira parte do processo consiste em identificar qual o registo a carregar. A segunda parte consiste em especificar o número a carregar no registo escolhido. Isto significa identificar A, B, C (ou outro registo) e indicar o número a colocar nele. A representação em linguagem assembly da instrução para carregar no registo A o número 11 teria a seguinte aparência: LD A,0B (0B hexadecimal). No entanto, antes de isto poder ser executado seria necessário recorrer a um programa Assembler para indicar ao computador o que se pretende que faça. Se estiver a ser usado um programa carregador — o que recomendo ao leitor a fim de se aperceber do que faz — seria necessário dar entrada primeiro ao número 3E. Este diz ao computador que deve carregar um número no registo A. A este número deve seguir-se 0B, e a máquina "compreenderá" que é o número a carregar no registo A.

Nota: É essencial recordar que sempre que é dado um código de uma instrução ou uma ordem em linguagem assembly e existem dois X's, quatro X's ou as letras "dis" depois da ordem, isto significa que o leitor deverá substituí-los por um ou dois bytes (valores) escolhidos por si. Por exemplo, quando se executa a ordem LD A,XX, os dois X's significam que o leitor deve introduzir um número de um só byte à frente de LD A, em substituição dos X's.

Podemos utilizar este método para carregar facilmente um número em qualquer dos registos A, B, C, D, E, ..., mas não devemos esquecer que é necessário usar um código diferente ou uma mnemónica diferente para cada um dos registos. Antes de continuarmos vale a pena ver como isto funciona. É necessário manter presentes dois aspectos importantes (se bem que não seja crucial entendê-los por agora) para podermos criar e usar um programa em código-máquina. O primeiro é que no final de um programa em código-máquina é necessário acrescentar a ordem RET (retorno). RET funciona exactamente do mesmo modo que o retorno de uma rotina GOSUB em Basic. Por exemplo, se fizer GOSUB 1000 num programa Basic, e esta subrotina terminar na própria linha 1000, o leitor escreve RETURN no final da linha para indicar ao computador que deve voltar para o ponto do programa onde se encontrava antes, ou mais exactamente para a instrução que se segue a GOSUB 1000. Se utilizar uma rotina em código-máquina, chamada por um programa Basic através da função USR, que significa User Subroutine (Subrotina do utilizador), deve incluir uma instrução RET no final do código-máquina para que o computador volte ao programa Basic. Dentro em pouco veremos porque razão devemos fazê-lo, mas para já convém que o leitor tenha presente esta necessidade. O código--máquina que corresponde à instrução mnemónica RET é C9 (hexa). O segundo ponto a ter em conta é que no caso de aceder a um programa em código-máquina através da ordem PRINT USR xxxxx, o número que é impresso quando a subrotina do utilizador é terminada corresponde ao conteúdo do par de registos BC. Este facto pode ser-nos extremamente útil.

O programa que se segue carrega muito simplesmente 0 nos registos B e C e em seguida volta ao programa Basic, imprimindo 0 ao fazê-lo. Para introduzir este programa no computador use o carregador hexadecimal, e escreva os números indicados. Depois de todo o programa ter entrado, não tente executá-lo com RUN... Escreva STOP e em seguida PRINT USR 30000.Deste modo a rotina em código máquina voltará ao programa Basic, mas imprimindo simultaneamente o conteúdo do par de registos BC. Depois de ter feito isto experimente alterar o programa colocando diferentes valores nos registos B e C, verificando como é impressa a resposta. Não se esqueça de que está a usar um número de dois bytes, com um máximo de 255 em cada um destes.

Código de operaração	Hexadecimal
LD B,00	0600
LD C,00	0E00
RET	C9

É lógico passarmos da carga de números em registos à carga do conteúdo de um registo noutro. Necessitamos uma vez mais de diferentes códigos para cada uma destas operações (por exemplo carregar no registo A o conteúdo do registo B), o que torna necessário um grande número de códigos diferentes, pelo que vamos por enquanto concentrar-nos na carga do conteúdo dos diversos registos apenas no registo A. O princípio aplicado na carga de qualquer outro registo é rigorosamente igual, sendo apenas diferentes os códigos usados. Basta compreender um para compreender todos.

Se quisermos carregar no registo A o conteúdo do registo B, conseguiremos fazê-lo utilizando a instrução simples LD A,B (significando carregar B em A). Se voltarmos à nossa anterior analogia entre o computador e uma pessoa vestindo um casaco, poderemos pensar nesta operação do seguinte modo:

1. Abrir a algibeira A, tirar o que nela se encontra e deitar fora.
2. Abrir a algibeira B, descobrir quantos artigos se encontram nela, mas não os tirar.
3. Arranjar o mesmo número de artigos que existem em B e guardá-los na algibeira A.

Existem agora na algibeira A tantos artigos quantos os presentes em B. É importante notar que o valor inicial de A nada tem a ver com o resultado desta operação.

Pode-se carregar o conteúdo de qualquer outro registo no registo A. As mnemónicas e códigos de linguagem-máquina para estas instruções são indicados a seguir:

Código de operação	Hexadecimal	Decimal
LD A,A	7F	127
LD A,B	78	120
LD A,C	79	121
LD A,D	7A	122
LD A,E	7B	123
LD A,H	7C	124
LD A,L	7D	125

A seguir o leitor deve procurar as mnemónicas e os códigos-máquina das instruções que carregam o conteúdo dos vários registos nos registos B e C, usando-as de tal modo que a resposta seja impressa no visor quando a máquina volta ao comando Basic.

... MAS AINDA SERVEM PARA FAZER CONTAS!

V

COMO FAZER SOMAS NOS REGISTOS

Agora que sabemos o que os registos são e o modo de colocar números neles, quer carregando-os directamente quer passando-os de um registo para o outro, é necessário discutir o modo de manipular estes registos. Se já programou em Basic sabe que é praticamente inútil atribuir um valor a uma variável se não for possível alterá-la depois aritmeticamente. Isto significa que é necessário "trabalhar" sobre um número, somando ou subtraindo a partir dele. É isso que estudaremos neste capítulo. Veremos igualmente o que acontece quando um número é demasiado grande ou pequeno para ser manipulado num registo. Isto conduzir-nos-á a explorar a "flag" (bandeira) de transporte ("Carry Flag") e os elementos de informação que lhe estão associados no interior do computador.

O leitor já terá provavelmente começado a pensar em como será possível somar dois registos, e se é possível fazê-lo exactamente do mesmo modo que em Basic, onde basta escrever: LET AB=AB+XY. O resultado da operação ficaria assim contido em AB. Felizmente é possível realizar uma operação igualmente simples em código-máquina, mas a instrução dada é obviamente diferente. Se queremos somar o conteúdo do par de registos DE ao conteúdo do par de registos HL, em vez de escrever LET HL=HL+DE (o que faríamos em Basic) é necessário usar ADD HL,DE. O exemplo abaixo mostra exactamente o que aconteceria se utilizássemos esta instrução num programa. Vale a pena notar que as primeiras duas linhas contêm os dois números a somar; a terceira realiza a operação propriamente dita, e as linhas 4 e 5 carregam o resultado no registo BC. Isto deve ser feito usando duas ordens diferentes — somando os dois registos — porque não existe nenhuma instrução do tipo LD BC,HL; a última ordem, de RETorno, limita-se a devolver o comando à Basic. Recomendo ao leitor que experimente imediatamente este programa:

```
ORG 30000
30000 21 01 00   LD HL,0001
30003 11 00 80   LD DE,32676
30006 19         ADD HL,DE
30007 44         LD B,H
30008 4D         LD C,L
30009 C9         RET
```

É vantajoso que o número máximo que pode ser contido num par de registos seja superior a 65000, pois é normalmente improvável que a soma de dois números seja superior a isto. No entanto, é útil saber o que acontecerá se a soma for superior. O pequeno programa que se segue esclarecerá o assunto. Experimente-o e descubra a resposta...

```
ORG 30000
30000 21 01 00   ld hl,0001
30003 11 FF FF   ld de,65535
30006 19         add hl,de
30007 44         ld b,h
30008 4D         ld c,l
30009 C9         ret
```

Antes de lhe dizer a resposta, observemos rapidamente este exemplo. A primeira ordem carrega 1 no par de registos HL. "Sim", dirá o leitor, mas o número depois de 21 no Código de Operação não é 1, é 01 00. Encontramos assim um aspecto curioso que convirá esclarecer antes de prosseguirmos. Quando carregamos um número num par de registos, em vez de colocarmos o byte mais significativo antes do menos significativo, fazemos o inverso. Portanto, para carregar o registo HL com 1, é necessário escrever 01 00. A parte mais significativa (o zero-zero) segue-se à menos significativa (ou um). Se o leitor não recorda o que significa "mais significativo" talvez convenha voltar ao capítulo deste livro onde o assunto foi referido. Para consolidar esta ideia, vejamos algumas rotinas rápidas que utilizam este mé-

todo. Felizmente, a carga do byte mais significativo depois do menos significativo não é invulgar num programa em código-máquina, pelo que acabará por tornar-se habitual ao leitor. Experimentemos as rotinas seguintes.

1.

```
org 30000
30000 01 00 01   ld bc,0100
30003 C9         ret
```

A resposta pretendida é 1, mas neste primeiro exemplo carregámos este valor no par de registos BC trocando os bytes mais e menos significativo. Nestas condições, o computador devolve-nos a resposta "256"... Quando se escreve em linguagem assembly é normal escrever o número em hexadecimal, e pela ordem correcta, ou seja, com o byte mais significativo antes do menos significativo. Como o leitor pode verificar neste primeiro exemplo, quando se trata de converter para código hexadecimal o byte mais significativo é novamente colocado antes do menos significativo, o que está errado.

2.

```
org 30000
30000 01 01 00   ld bc,0001
30003 C9         ret
```

Desta vez colocou-se o byte mais significativo depois do menos significativo na listagem em hexadecimal, pelo que obtemos a resposta correcta, 1.

3.

```
org 30000
30000 01 FA 7B   ld bc,64123
30003 C9         ret
```

Neste exemplo verificamos que os bytes foram novamente invertidos. A resposta é portanto incorrecta. Mas antes de passarmos à rotina seguinte tentemos determinar, em decimal, qual o resultado desta.

4.

```
org 30000
30000 01 7B FA   ld bc,64123
30003 C9         ret
```

Antes de voltarmos ao assunto que estávamos a tratar, recapitulemos; podemos dizer que, ao carregarmos um número num par de registos, o byte mais significativo se segue ao menos significativo em codificação-máquina. Este é um outro exemplo da utilidade que pode ter um programa Assembler, ao qual indicamos o valor na sua configuração normal, ou seja começando pelo byte mais significativo.

Voltemos ao exemplo anterior. Já dissemos que a primeira ordem carrega 1 no par de registos HL. A ordem seguinte carrega FFFF (hexadecimal) no par de registos DE, o que, como o leitor certamente recorda, é equivalente a 65535 em decimal (que por sua vez é o maior número que se pode guardar num par de registos). Se somarmos agora 1, ultrapassa-se este valor máximo. Quando tal acontece em Basic, a máquina imprime uma mensagem de erro e o programa pára. Em código-máquina isto não acontece, porque não existem códigos de erro; acontece simplesmente um erro! Neste caso particular, em vez de se obter 65536, um valor que está para além da gama possível, os registos passam ao valor zero. Por outro lado, quando ocorre este excesso, o computador recorda o facto passando a "carry flag" ao valor 1. Não se preocupe por agora com o que esta "carry flag" possa ser; voltaremos ao assunto dentro em pouco.

É necessário recordar duas questões importantes quanto à soma de dois pares de registos. Em primeiro lugar, só se pode somar um qualquer par de registos a HL. Por outro lado, só se podem somar pares de registos a outros pares de registos. Não se pode somar um registo simples a um par de registos, ou vice-versa. Mais adiante apresenta-se uma lista dos códigos hexadecimais das operações de soma de pares de registos. Talvez o leitor se interrogue sobre o porquê da existência de uma instrução para somar HL a si mesmo, mas trata-se de facto de um modo muito simples de duplicar o valor de HL. Também vale a pena notar que cada uma destas ordens ocupa um único byte. Compare-se isto com o equivalente Basic, que consome cerca de dez vezes mais memória!

```
org 30000
30000 00        ld hl,bc
30001 19        ldhl,de
30002 29        ldhl,hl
```

Estudemos agora a soma de registos simples. Recordando que os registos simples, de um só byte, apenas podem conter números na gama 0 a 255, a possibilidade de esta gama ser excedida é muito mais provável do que no caso de pares de registos. O que se passa quando tal acontece? Bom, passam simplesmente a zero, tal como no caso dos pares de registos. O problema é: podemos fazer qualquer coisa quanto a isto? Sim... sempre que somamos dois números, pode ou não haver um excesso ("carry"). O computador possui um registo muito especial para tratar questões deste tipo. É conhecido pelo nome de registo F ("F" de *flags*). Não é habitual usarmos directamente este registo, muito simplesmente porque o computador o monopoliza para poder tratar pequenos elementos de informação; separa cada um dos bits que o constituem, usando-os depois independentemente. Ao ler este livro o leitor encontrará muitas referências a "flags" (bandeiras) que se encontram "set" ou "reset"; "set" significa que se encontram ao nível lógico 1, e "reset" ao nível lógico 0. Uma destas flags é a já nossa conhecida "carry flag". O aspecto mais útil quanto a esta flag é que sempre que se somam dois registos simples e a resposta é superior a 255, a flag em causa é passada ao valor 1. Se tal não acontecer a flag é limpa (passando a 0). Não podemos aceder o registo F directamente, e portanto não é possível alterar as flags de um modo directo, se bem que seja de facto relativamente simples actuar sobre elas.

É possível somar qualquer registo ao registo A, incluindo ele próprio (se bem que não seja possível somar qualquer registo a outro que não seja o A). O leitor descobrirá aliás que este registo A, muitas vezes designado por "Acumulador", possui capacidades muito mais extensas do que os outros. Não existe qualquer razão especial para isto, excepto tratar-se da primeira letra do alfabeto — e da inicial da palavra Acumulador — sendo portanto fácil recordar que é ele que se encontra nestas condições. Muitas das instruções importantes usam este registo, tal como, entre os pares de registos, o mais usado é o HL. Do mesmo modo que se pode somar qualquer registo ao registo A, também se pode somar um número directamente a este registo, mas convém não esquecer que o número em causa deve variar entre 0 e 255. Em seguida apresentamos uma lista das diferentes ordens para somar (em inglês, "add") e dos códigos-máquina que lhes equivalem. Estes códigos seguem-se uns aos outros, pelo que é razoavelmente fácil recordá-los.

Código de operação	Hexadecimal	Decimal
ADD A,(HL)	86	134
ADD A,A	87	135
ADD A,B	80	128
ADD A,C	81	129
ADD A,D	82	130
ADD A,E	83	131
ADD A,H	84	132
ADD A,L	85	133
ADD A,xx	C6xx	198,xx

A soma de dois registos simples é muito semelhante à de pares de registos, como se mostra nos exemplos seguintes:

Exemplo 1:

```
org 30000
30000 3E 05     ld a,5
30002 06 06     ld b,6
30004 80        add a,b
30005 4F        ld c,a
30006 06 00     ld b,0
30008 C9        ret
```

Exemplo 2:

```
org 30000
30000 3E 0E     ld a,14
30002 C6 06     add a,06
30004 4F        ld c,a
30005 06 00     ld b,0
30007 C9        ret
```

No exemplo 1 as duas primeiras instruções carregam simplesmente 5 e 6 nos registos A e B respectivamente. A ordem 3 soma

o conteúdo do registo B ao conteúdo do registo A, mantendo o resultado neste último registo. As duas linhas seguintes tornam-nos possível imprimir a resposta usando a simples ordem PRINT USR da Basic.

No segundo exemplo somamos simplesmente 6 ao conteúdo do acumulador ou registo A, antes de carregarmos o conteúdo de A no registo C e carregarmos B com 0, executando retorno em seguida de tal modo que o número apresentado depois da ordem PRINT USR seja o conteúdo de BC — ou por outras palavras, a resposta a A + 6. Uma instrução ADD actuará sempre sobre a flag "carry"; como se disse já, se não houver transporte esta flag passará a zero; se houver, assumirá o valor um.

Até agora não conseguimos utilizar este resultado de nenhuma forma. O modo mais simples de o utilizarmos, porém, consiste em recorrer à ordem ADC. Esta ordem significa "somar com transporte", funcionando precisamente do modo descrito. Suponhamos que o computador encontra uma instrução ADC A,B. Lerá o conteúdo do registo B, somá-lo-á ao conteúdo do registo A, e deixará a resposta no registo A tal como na instrução anterior, ADD A,B. Além disso somará a flag "carry" ao novo número. Como esta ainda não foi accionada por esta instrução, o número que é somado ao resultado corresponde ao estado anterior da flag. O resultado final é armazenado no registo A, actuando finalmente sobre a flag. Portanto, se esta tiver sido passada a 1 por uma soma anterior, e não tiver sido alterada desde então, o resultado da ordem ADC será afectado. À primeira vista isto pode parecer mais uma desvantagem, do que um benefício, mas se o leitor se recordar dos seus tempos de escola, quando aprendeu a somar, depressa compreenderá que se trata de um processo lógico e extremamente útil. Corresponde simplesmente ao "e vai um" que usámos na escola para realizar as somas. Recordamos que começávamos por somar a coluna do lado direito, dizendo "e vai um" quando o resultado dessa coluna ultrapassava 9 e se tornava necessário somar 1 à coluna seguinte. Por exemplo, se somarmos os números 14 e 7, somamos primeiro 4 e 7, sendo o resultado 11, pelo que escrevemos "1" ao resultado da coluna da esquerda, obtendo neste caso o resultado 2. Lemos em seguida o resultado da esquerda para a direita, obtendo "21".

O exemplo abaixo mostra como podemos utilizar esta função num programa em código-máquina. Como o programa é razoavelmente comprido, e um pouco complicado, talvez seja melhor que o leitor não se preocupe com a sua explicação até o ter escrito e verificado como funciona — o que explicarei no parágrafo seguinte. Por enquanto preocupe-se apenas em escrever exactamente o que está escrito nesta página.

```
org 30000
30000 16 33        ld d,51
30002 1E 85        ld e,133
30004 26 7B        ld h,123
30006 2E 07        ld l,199
30008 7D           ld a,l
30009 83           add, a,e
30010 6F           ld l,a
30011 7C           ld a,h
30012 8A           adc a,d
30013 67           ld h,a
30014 44           ld b,h
30015 4D           ld c,l
30016 C9           ret
```

A ordem 1 atribui o valor 51 ao registo D. A ordem 2 atribui o valor 133 ao registo E. A ordem 3 atribui o valor 123 ao registo H e a ordem 4 o valor 199 a L. O objectivo deste programa consiste em somar o conteúdo do par de registos DE ao par de registos HL. Isto não é feito utilizando uma ordem única, mas sim somando as duas metades de cada par. Por outras palavras, queremos apenas somar E a L, e em seguida D a H. O primeiro problema que encontramos é o facto de só podermos somar um registo ao registo A, sendo portanto necessário substituir o registo L pelo registo A. Isto é fácil recorrendo à ordem 5, que carrega em A o conteúdo do registo L. Depois disto podemos somar E a A. O valor contido em A é agora igual à soma do conteúdo de L e do conteúdo de E, mas nós queremos que este resultado esteja em L, e não em A. Vencemos este problema muito simplesmente carregando novamente o valor de A em L.

Como já somámos os dois bytes menos significativos, usando a ordem 6, a flag "carry" passou necessariamente ao valor um se houve um excesso — o que de facto aconteceu. Repetimos agora o procedimento para somar os dois bytes mais significativos, ou, seja, somar D a H. Desta vez, no entanto, usamos a or-

dem ADC — e somamos com transporte. Acontece portanto que somamos D a A, do mesmo modo que anteriormente, mas acrescentamos ao resultado o conteúdo da flag "carry". Assim, se quando somámos L e E o resultado foi superior a 255, estaremos agora a acrescentar 1 ao valor dos dois bytes mais significativos. Obtemos assim um resultado rigoroso. As ordens 11 e 12 são apenas as instruções já nossas conhecidas que nos permitem carregar o conteúdo de HL em BC, de modo a imprimir o resultado no visor utilizando a instrução Basic PRINT USR. Um outro aspecto a notar neste programa é que as duas ordens que se encontram depois de ADD A,E mas antes de ADC A,D não afectam de modo algum o estado da flag "carry". Se o leitor pretende usar este tipo de instrução é útil saber quais as ordens que alteram o conteúdo da flag "carry", de modo a saber qual o resultado a esperar. Por esta razão, na parte final do livro poderá encontrar uma lista de todas as ordens e do modo como afectam ou não a flag "carry".

Neste momento o leitor já compreende certamente a diferença entre ADD e ADC. Vejamos agora os códigos e respectivas mnemónicas para as diferentes combinações.

Código de operação	Hexadecimal	Decimal
ADC A,(HL)	8E	142
ADC A,A	8F	143
ADC A,B	88	136
ADC A,C	89	137
ADC A,D	8A	138
ADC A,E	8B	139
ADC A,H	8C	140
ADC A,L	8D	141
ADC A,xx	CExx	206,xx
ADC HL,BC	ED4A	237,74
ADC HL,DE	ED5A	237,90
ADC HL,HL	ED6A	237,106

Não é possível somar directamente uma constante a um par de registos, mas é possível consegui-lo facilmente carregando num par de registos o número que se deseja somar a HL e somando em seguida o outro par de registos a HL. O resultado em HL será igual ao que continha anteriormente mais o novo número. Por exemplo:

```
org 30000
30000 11 39 00          ld de,57
30003 19                add hl,de
```

Este método tem a desvantagem de requerer o uso do par de registos DE, que o leitor pode querer utilizar para outras operações. Um outro modo de conseguir o mesmo objectivo é o apresentado em seguida, mas desta vez o único registo que é alterado, além do HL, é o registo A. Note que neste exemplo usei novamente a ordem "somar com transporte", tal como a usei no primeiro exemplo — de tal modo que no caso de o byte menos significativo dar um excesso será somado um ao byte seguinte, tornando o resultado rigoroso.

```
org 30000
30000 7D                ld a,l
30001 C6 39             add a,57
30003 6F                ld l,a
30004 7C                ld a,h
30005 CE 00             adc a,0
30007 57                ld h,a
```

Imagino que o leitor já está um pouco farto de tratar este assunto em abstracto, de modo que vamos estudar um programa em código-máquina que talvez lhe seja útil. Se bem que possua algumas das ordens que já estudámos, não é essencial que o leitor o compreenda completamente nesta fase — não se preocupe portanto se ainda lhe parecer um tanto misterioso. Pode ter a certeza de que quando chegar ao final do livro poderá voltar atrás e compreendê-lo perfeitamente. O objectivo da apresentação deste programa consiste em permitir-lhe usá-lo, ver os seus efeitos, e descansar um pouco do estudo.

```
org 30000
30000 21 00 40          ld hl,16384
30003 06 C0             ld b,192
Alterar B e HL para diferentes blocos do visor
```

```
A
30005 0E 20        ld c,32
B
30007 5E           ld e,(hl)
30008 CB 1E        rr (hl)
30010 23           inc hl
30011 0D           dec c
30012 20 F9        jr nz,B
30014 CB 43        bit 0,e
30016 28 09        jr z,C
30018 E5           push hl
30019 11 1F 00     ld de,31
30022 ED 52        sbc hl,de
30024 CB FE        set 7,(hl)
Para não reintroduzir os pixels,RES 7,(HL)
30026 E1           pop hl
C
30027 A7           and a
30028 10 E7        djnz,A
30030 C9           ret
```

Como prelúdio à secção seguinte, observe esta listagem e note que algumas das ordens possuem parêntesis.

CARGA DO CONTEÚDO DE UMA POSIÇÃO DE MEMÓRIA NUM REGISTO

Mais atrás, como o leitor talvez recorde, observámos o modo como os números são guardados em caixas — ou posições de memória. Como se disse então, cada uma destas caixas possuía um número, indicativo do seu endereço único. A razão de ser deste endereço consiste em permitir-nos aceder com relativa facilidade ao conteúdo de cada caixa. Já referi a utilidade de podermos manipular à nossa vontade os números contidos nas caixas, mas por agora vê-mo-nos forçados a falar ainda do modo como é possível tirar números das caixas, ou posições de memória, e passá-los para os registos. Obviamente, quando esses números se encontrarem nos registos ser-nos-á possível manipulá-los. Por outro lado é possível carregar o conteúdo do acumulador ou de qualquer outro registo numa posição ou endereço de memória à

nossa escolha; deste modo torna-se possível guardar um dado valor para uso futuro. Note-se ainda que, usando este método, é possível aceder em qualquer momento a uma vasta quantidade de números.
Definamos primeiro o que de facto queremos fazer. A nossa tarefa consiste em carregar no acumulador (registo A), por exemplo, o conteúdo de uma determinada posição de memória. Esta posição terá um endereço. Se escrevermos simplesmente LD A e em seguida o endereço, o computador confundirá o endereço com um número que queremos carregar no registo. Se por exemplo escrevermos LD A,12, a máquina pensará que queremos é carregar no acumulador o conteúdo de uma determinada posição de memória. De facto é muito simples fazê-lo, dado que nos basta escrever o endereço dentro de parêntesis. Por exemplo, LD A,(2465). O computador carregará então o valor que se encontra na posição de memória 2465 no acumulador. Como cada posição só pode conter um número entre 0 e 255, não há qualquer dificuldade em carregar num registo simples o conteúdo de qualquer posição de memória. Se no endereço 2465 se encontrar o número 64, depois da execução desta instrução o acumulador conterá igualmente 64. Tal como acontece em todas estas ordens, o conteúdo do endereço é ainda mantido. Por outras palavras, o conteúdo do endereço 2465 é ainda 64. Esta ordem é muito útil, dado que nos permite obter o conteúdo de qualquer posição de memória, colocá-lo num registo, e manipulá-lo. Por exemplo:

```
org 30000
30000 3A 30 75     ld a,(30000)
30003 C6 29        add a,41
```

O único problema agora consiste em saber como colocar o número manipulado de novo no mesmo ou noutro endereço. Isto pode ser feito com bastante facilidade usando uma variante da ordem anterior. Se pensar naquilo que agora pretendemos fazer, compreenderá que em vez de carregar o acumulador com o conteúdo do seu endereço, nos interessa carregar no endereço o conteúdo do acumulador. Usamos portanto esta ordem invertida: LD (ADDR),A, sendo ADDR o endereço.

Apresentamos em seguida um exemplo do funcionamento deste conceito:

```
org 30000
30000 3A 30 75          ld a,(30000)
30003 C6 0F             add a,15
30005 32 30 75          ld (30000),a
30008 C9                ret
```

Voltando à nossa primeira ideia, a saber, retirar um número de uma posição de memória, manipulá-lo e em seguida voltar a colocá-lo nessa posição, poderemos usá-la para alterar por exemplo a cor do visor — partindo da cor que nele se encontra e manipulando-a até se transformar numa cor diferente. Como o visor, no que se refere à cor, possui um comprimento de 704 bytes, é necessário realizar esta operação 704 vezes. Felizmente é possível utilizar uma nova instrução que nos permite evitar a escrita do mesmo programa 704 vezes! Voltaremos a isto um pouco mais adiante. Por agora, antes de passarmos ao exemplo da alteração da cor do visor, proponho uma pequena tarefa ao leitor: escreva uma rotina que parta do código de qualquer caracter em memória. Vou dar uma sugestão: o inverso de cada caracter é 128 vezes maior do que o seu código normal (em decimal). Por outras palavras, se partir do código do caracter "a" e lhe somar 128, obterá o "a invertido".

```
org 30000
30000 21 00 58          ld hl,22528
30003 01 C0 62          ld bc,704
30006 1E                ld e,6
loop
30008 7E                ld a,(hl)
30009 CB 87             res 0,a
30011 CB 8F             res 1,a
30013 CB 97             res 2,a
30015 83                add a,e
30016 77                ld (hl),a
30017 23                inc hl
30018 0B                dec bc
30019 79                ld a,c
30020 FE 00             cp 0
30022 20 F0             jr nz,loop
30024 78                ld a,b
```



```
30025 FE 00             cp 0
30027 20 EB             jr nz,loop
30029 C9                ret
```

Nota: carrega-se em E a nova cor de tinta.

Subtracção

Em código-máquina, as instruções para subtrair números e registos de outro registo são exactamente iguais às suas equivalentes para a soma. Nestas condições, a subtracção é bastante fácil de compreender. Existem obviamente diferenças; no caso da soma pode-se ultrapassar o maior número que o registo pode conter, e no da subtracção pode-se ultrapassar o *menor* número — o zero. Por outras palavras, é possível que o número original seja menor do que aquele que lhe é diminuído. Experimente tirar 11 a 5 — obterá 6. Se ocorrer este excesso para menos, a flag "carry" é passada ao valor 1, e se não houver excesso a flag em causa passará a zero. Quando ocorre um excesso na soma, os números começam novamente a somar a partir de zero. Por exemplo, se se somar 5 a 255, o computador passará a 0 e somará quatro; mas na subtracção, a máquina passará a 256 e diminuirá em seguida.

A subtracção funciona do mesmo modo que a soma, o que significa que (a instrução SUB indica subtracção) diminuirá o valor do registo B do valor do registo A, armazenando o resultado neste último registo. A flag "carry" será passada a "1" ou "0" conforme o resultado.

Como a ordem SUB só se aplica a registos de um único byte, e aliás apenas ao acumulador, é vulgar (em vez de escrever SUB A,B) escrever apenas SUB B. Isto pode parecer de início um pouco confuso, mas o leitor habituar-se-á rapidamente. Como a subtracção não é um dos temas mais simples quando se usa um formato pouco familiar, talvez seja boa ideia neste ponto ver o que já foi dito sobre a subtracção mais atrás, a fim de facilitar a compreensão do tema pelo leitor. As diferentes ordens assembly, e os códigos-máquina correspondentes, são apresentados em seguida:

Código de operação	Hexadecimal	Decimal
SUB (HL)	96	150
SUB A	97	151
SUB B	90	144
SUB C	91	145
SUB D	92	146
SUB E	93	147
SUB H	94	148
SUB L	95	149
SUB xx	D6xx	214,xx

Apresentamos agora um exemplo sobre o modo de utilização das ordens de subtracção:

```
org 30000
30000 3E 0B        ld a,11
30002 06 07        ld b,7
30004 97           sub a,b
30005 4F           ld c,a
30006 06 00        ld b,0
30008 C9           ret
```

É possível subtrair constantes numéricas ao registo A. Por exemplo, a instrução SUB A,100 diminuirá 100 ao número armazenado no registo A. O resultado é então guardado no registo A. O leitor deve notar que apesar de existirem instruções ADD para pares de registos, não existem quaisquer ordens de subtracção para pares de registados. Em seguida apresentamos um curto exemplo de subtracção.

```
ORG 30000
30000 0600         LD B,00
30002 3E58         LD A,88
30004 D633         SUB 51
30006 4F           LD C,A
30007 C9           RET
```

A subtracção com transporte (SBC), por outro lado, funcionará para pares de registos; mas tal como acontece no caso das ordens ADD e ADC só o valor do par de registos HL pode ser alterado, ou o conteúdo do registo simples A. SBC A, C subtrairá o valor do registo C ao valor do registo A, subtraindo em seguida ao resultado o valor da flag "carry" e guardando o resultado final no registo A. A ordem "subtrair com transporte" pode também ser usada para subtrair dois números um do outro quando pode haver um excesso.

Pensando um pouco na forma como aprendemos originalmente a subtrair números podemos compreender como se realiza esta operação no computador. Se quisermos tirar 19 a 21, actuaremos do seguinte modo. Primeiro tiramos 9 a 1. Isto não é exactamente possível, pelo que somos obrigados a ir buscar 1 (ou seja, 10) à coluna da esquerda. Passamos portanto a tirar 9 de 11, o que dá como resultado "2". Temos de ter agora em conta que a coluna seguinte foi reduzida de 1, produzindo um excesso para menos; diminuímos portanto 1 de 2, obtendo o resultado 1 — mas como temos de tirar igualmente o 1 presente na flag "carry" o resultado nesta coluna é zero, e o resultado final é 2.

É fácil compreender, a partir da lista que se segue, que o uso de "subtracção com transporte" sobre um par de registos requer dois bytes, e que uma operação sobre um registo simples requer apenas um byte — o que torna estas ordens bastante económicas. Os códigos assembly e máquina são listados em seguida:

Código de operação	Hexadecimal	Decimal
SBC A,(HL)	9E	158
SBC A,A	9F	159
SBC A,B	98	152
SBC A,C	99	153
SBC A,D	9A	154
SBC A,E	9B	155
SBC A,H	9C	156
SBC A,L	9D	157
SBC A,xx	DExx	222,xx
SBC HL,BC	ED42	237,66
SBC HL,DE	ED52	237,82
SBC HL,HL	ED62	237,98

Operações sobre a flag "carry"

É muitas vezes útil, quando se usam instruções como a Sub-

tracção ou a Soma com Transporte, poder determinar o conteúdo da flag "carry". Infelizmente não é possível limpar esta flag, ou seja passá-la a 0, mas é possível consegui-lo recorrendo a um truque. Sabemos que sempre que se soma um número ao acumulador, a flag em causa é passada a 1 ou a 0 conforme há ou não transporte. Se somarmos 0 ao acumulador não ocorre evidentemente qualquer transporte — e conseguimos portanto passar a flag "carry" para o valor zero. Esta é provavelmente a ordem mais útil envolvendo esta flag. Quando se usam as ordens mencionadas, pode ser particularmente grave que a flag esteja ao valor 1 quando devia estar a zero. Por outro lado, passar a flag "carry" para 1 é muito mais fácil, existindo uma instrução precisamente para este efeito. A instrução em causa é a SCF — "Set Carry Flag" (passar a 1 a flag "carry"). O código-máquina para esta operação é 37 (hexadecimal).

Terminamos assim este capítulo. Estudámos duas formas muito importantes de manipular os registos, e os modos de retirar números de posições de memória e passá-los para registos. Estudámos igualmente ordens complicadas — como a soma com transporte — e os modos de vencer certos problemas actuando sobre a flag "carry". Espero que o leitor já seja capaz de escrever um pequeno programa usando estas ordens.

VI

ASSEMBLERS, DISASSEMBLERS E PROGRAMAS DE DETERMINAÇÃO DE ERROS

Neste capítulo faremos um intervalo na aprendizagem de código-máquina. Em vez de continuarmos o nosso estudo, iremos observar os três principais tipos de programas escritos especificamente para auxiliar aqueles que programam em código-máquina. Tendo já adquirido alguns conhecimentos básicos sobre a escrita de programas simples em linguagem-máquina, o leitor estará dentro em pouco pronto a lançar-se em maiores aventuras no campo da programação; isto conduzi-lo-á à escrita de rotinas relativamente complexas em código-máquina, e por esta razão ser-lhe-á útil dispor de alguns auxiliares que tornem o seu trabalho mais fácil. A maior parte destes auxiliares são aliás escritos em código-máquina. Como é óbvio, o programador que os escreveu será alguém com um bom conhecimento do assunto, e não tenderá a definir os modos de execução da forma mais simples. Como consequência, quando se tratar de decifrar os manuais de instrução fornecidos juntamente com programas Assembler ou "Debugger" (programas usados para determinar erros de programação), o principiante terá alguma dificuldade em compreender certas passagens. Por esta razão incluí no final deste capítulo uma curta secção explicando exactamente o modo de usar estes programas. É igualmente útil notar que as minhas observações se referirão a programas existentes comercialmente.

A escolha de um programa comercial depende obviamente das preferências e da experiência de cada um de nós. Para esclarecimento do leitor, posso no entanto indicar que todos os programas deste livro foram traduzidos para código-máquina ("assembled") ou deste código para linguagem Assembly ("disassembled") usando programas da ACS Software.

Assemblers

Neste momento, é importante sabermos exactamente o que são as linguagens Assembly e as linguagens-máquina. Vamos portanto recordar o que já foi dito sobre o assunto (se o leitor pensa já ter uma ideia suficiente sobre este assunto pode passar à secção seguinte).

A "linguagem-máquina" é constituída apenas por números. O computador só compreende números, sendo apenas capaz de executar instruções que lhe sejam dadas dessa forma. Infelizmente, recordar uma grande quantidade de números e respectivos significados não é uma tarefa particularmente fácil para os seres humanos. Por estas razões foi desenvolvida uma "linguagem Assembly". Cada instrução da Assembley corresponde exactamente a uma ordem em código-máquina, consistindo a única diferença em as instruções de linguagem Assembly serem escritas usando mnemónicas que indicam imediatamente o objectivo de cada uma delas. Por exemplo: se quisermos executar a ordem "Carregar no registo A o conteúdo do registo B", podemos conseguir este resultado escrevendo em código-máquina o número 78. Mas não nos é fácil recordar que este número 78, num total de cerca de 600 números, corresponde à função requerida. Um modo de evitar este problema consistirá em dispor de uma lista de todas as instruções e códigos correspondentes, que nos permita descobrir aquele de que necessitamos.

O primeiro problema disto é que ocupa muito espaço incluir na lista de instruções todas as explicações necessárias. Prefere-se portanto indicar para cada instrução uma mnemónica suficientemente fácil de compreender. Uma mnemónica é simplesmente uma abreviatura da ordem correspondente. Em vez de escrever "Carregar no registo A o conteúdo do registo B" basta-nos escrever a simples abreviatura LD A,B. O leitor já descobriu certamente que esta abreviatura é muito mais fácil de recordar do que o código numérico correspondente, e que ocupa muito menos espaço do que uma explicação completa. Se agora aprendermos todas as instruções em mnemónica obteremos aquilo a que se chama linguagem Assembly. Por outras palavras, esta linguagem é simplesmente formada pelas mnemónicas da linguagem-máquina. Nestas condições, compreende-se que a linguagem Assembly não é uma adaptação da linguagem-máquina ao contrário do que acontece com todas as linguagens de alto nível (como a Basic), antes consistindo num conjunto de mnemónicas univocamente correspondentes às instruções da linguagem-máquina.

Estamos portanto perante uma linguagem que é fácil de compreender e de usar em listagens convenientemente construídas. No entanto, apesar de ser uma linguagem bastante prática, requer ainda algum esforço para converter em código-máquina. Porque razão não utilizarmos então o nosso computador para realizar esta conversão? É isto que um programa Assembler pode fazer por nós: trata-se de um programa que transforma mnemónicas em código-máquina.

Um outro modo de pensar no funcionamento de um programa Assembler consiste em compará-lo com um restaurante. Quando queremos comer, pedimos um prato indicado num menú. Ao fazermos o nosso pedido o empregado escreve o prato desejado, mas muitas vezes identifica-o por um número, mais fácil e rápido de escrever. O empregado transforma portanto o nome do prato num número, tal como o programa Assembler transforma a mnemónica num número.

Todos os programas Assembler para o Spectrum funcionam do mesmo modo: primeiro escrevemos um programa usando mnemónicas, ou seja a linguagem Assembly, e quando este se encontra terminado instruímos o programa no sentido de o transformar em código-máquina. Este código é em seguida colocado numa determinada zona da memória, escolhida por nós. A sua posição depende igualmente da quantidade de memória de que o computador dispõe. Um programa Assembler é um programa relativamente complicado, e alguns dos existentes no mercado só são utilizáveis de facto no Spectrum de 48 K. Isto influenciou a minha escolha do Assembler para os programas deste livro, dado que os programas ACS funcionam em 16 e em 48 K.

Se o leitor ler alguma revista de micro-computadores, notará vários anúncios de Assemblers. O preço destes programas é normalmente um pouco mais elevado do que os programas de jogos. Isto acontece por duas razões: em primeiro lugar são, na sua maior parte, mais complicados do que os jogos; em segundo lugar têm um mercado mais reduzido e não são portanto vendidos em quantidades tão grandes. Acredite, porém, que merecem o dinheiro neles gasto. São extremamente úteis para quem quiser programar em código-máquina.

Disassemblers

Um programa Disassembler faz exactamente o contrário de um Assembler; converte o código-máquina em linguagem Assembly.

Pode-se usar um Disassembler de duas maneiras diferentes:

1 — Como um simples auxiliar para verificar se foi escrito na memória do computador exactamente aquilo que se queria.

2 — Estudar o que outros escreveram em código-máquina, e o modo como resolveram certos problemas.

Por exemplo, se o leitor quiser estudar certas partes da ROM Sinclair para compreender melhor o sistema operativo do Spectrum e se possível utilizar as rotinas nela contidas, poderá usar o Disassembler. Por outro lado, em muitas revistas, quando se apresentam listagens de código-máquina, estas surgem apenas em hexadecimal. Aconselho o leitor a carregar alguns programas de outras pessoas e "disassemblá-los", obtendo assim uma ideia do modo como os outros os escrevem. Pode ganhar muito obervando o uso das várias instruções por diferentes programadores.

No caso do Assembler e Disassembler usados neste livro, é possível carregá-los simultaneamente no Spectrum de 48 K. Quando adquirir um programa auxiliar, tenha em conta a utilidade deste tipo de "compatibilidade". Não é possível carregar um Assembler e um Disassembler numa máquina de 16 K simultaneamente porque esta não dispõe de memória suficiente. No entanto, nessas circunstâncias é possível carregar um deles de cada vez. Como é possível carregar e passar para cassette os seus próprios programas em código máquina independentemente dos programas que se encontrem já no seu computador, esses seus programas podem ser passados para código usando o Assembler, passados para cassette, e carregados novamente junto com um Disassembler para estudo.

Determinação de erros em programas

O terceiro e último tipo de programa que iremos considerar não será usado ao escrever programas mas é bastante útil para descobrir erros e fazer alterações ao programa depois de ter sido passado para linguagem-máquina. Quando se adquire um programa "Debugger", como é chamado, não se compra geralmente um programa único. Um bom programa oferecerá algumas das seguintes possibilidades:

1. A opção de fazer executar a sua rotina em código-máquina uma instrução de cada vez, imprimindo o resultado de todos os registos em seguida. Isto é particularmente útil quando é necessário localizar um erro num programa, ou quando se deseja saber exactamente como funciona a rotina de outra pessoa.

2. A possibilidade de inserir um "ponto final" em qualquer ponto do programa, ou seja, executar o programa até esse ponto e fazê-lo parar. Esta ordem pode ser útil de diversas maneiras: o leitor pode não estar interessado em conhecer o estado dos registos depois de cada instrução, em particular se se tratar de um programa comprido, mas quer conhecer o estado dos registos e das flags depois de ter sido executada uma certa operação ou alguns blocos de instruções. A execução do programa deve poder ser continuada facilmente, carregando apenas numa tecla.

3. Permitir alterar o estado de um dado registo quando o programa é parado, o que em geral pode ser feito com bastante facilidade. Isto pode ser útil em dois casos especiais. Em primeiro lugar, se quiser saber qual será o resultado se carregar em determinados registos certos valores. Em segundo lugar, pode ser útil ter possibilidades de simular determinadas condições em que é dada entrada a valores demasiado grandes e não se sabe se as rotinas para tratamento de erros dão os resultados desejados.

4. A possibilidade de executar o seu programa. Isto pode parecer óbvio, mas quando se está a executar um programa monitor (ou "debugger") nem sempre é fácil voltar à Basic e usar a ordem USR.

5. A possibilidade de converter números de hexadecimal para decimal, no interior do programa monitor.

6. Três ordens muito úteis que, quando usadas em conjunto, permitem ao monitor alterar o programa em código-máquina. Essas ordens permitem inspeccionar o conteúdo de um dado endereço, e alterá-lo em caso de necessidade. Além disso, no caso de muitos Assemblers, se se mantiver o dedo pousado sobre a tecla Enter, o conteúdo dos endereços subsequentes é automaticamente impresso no visor, mas não é aceite nenhum novo número a menos que o escrevamos, obviamente antes de carregar em Enter. Se o leitor resolver inserir uma ordem no meio de um programa em código-máquina, é muitas vezes difícil fazê-lo porque,

como é óbvio, isto obriga a deslocar o restante do código. No caso de muitos monitores é fácil conseguir o mesmo resultado indicando ao computador onde se deseja inserir a nova ordem, e a quantidade de bytes que desejamos inserir; o programa cria então o espaço necessário para a nova ordem. A instrução complementar a esta é evidentemente a instrução "delete" (apagar), que pode ser encontrada em muitos programas monitores e funciona exactamente ao contrário da ordem "insert".

O que já se disse constitui apenas um simples guia quanto ao que poderemos encontrar num programa monitor, se bem que geralmente este contenha algumas ordens extra para além das já citadas. O modo como os programas monitores funcionam é variável.

Muitas vezes pode-se misturar um Assembler ou Disassembler com um Monitor. Por exemplo, descobre-se por vezes num programa monitor a existência de uma função "disassembler". Um outro exemplo disto é o Assembler comercializado pela Picturesque. Além de passar as instruções para código, permite-nos igualmente montá-las usando diversas ordens sofisticadas. O programa monitor, que é basicamente um programa de determinação de erros e um Assembler, pode ser usado ao mesmo tempo que o programa Assembler/de montagem, formando assim uma combinação poderosa. Por uma questão de simplicidade, e para diminuir os preços, os programas Assembler/Disassembler/Monitor usados neste livro foram programas separados, existentes em cassettes diferentes.

Um programa simples de determinação de erros não deve necessariamente ser escrito em código-máquina. O leitor pode decidir não comprar um programa comercial nesta sua fase de aprendizagem. O melhor compromisso será então usar o programa monitor apresentado nas páginas que se seguem. Permitir-lhe-á introduzir código-máquina no computador e montá-lo satisfatoriamente. A conversão para mnemónicas pode ser realizada usando as tabelas de referência no final do livro.

Quando o leitor tiver introduzido na máquina este programa Basic não se esqueça de gravá-lo em cassette antes de o testar, e quando tiver a certeza de que o programa funciona convenientemente faça uma segunda cópia para o caso de a primeira se deteriorar por qualquer razão.


```
   5 CLEAR 31999
  10 PRINT "       ****Spectrum Monitor****" ""          by
James Walsh "
  30 REM carregar m/c
 100 INPUT "Endereço inicial (dec)?"; loc
 110 PRINT AT 5,0;
 120 PRINT loc; "...";
 130 LET n = PEEK loc: GO SUB 1400
 140 INPUT z$
 145 PRINT "....";z$
 150 IF z$ = "fim" OR z$ = "FIM" THEN GO TO 300
 160 IF z$ = "p" OR z$ = "P" THEN COPY : GO TO 120
 170 IF z$ = "S" OR z$ = "s" THEN GO TO 280
 190 IF z$ = "v" OR z$ = "V" THEN GO SUB 1600: GO TO
120
 200 IF z$ = "j" OR z$ = "J" THEN LET loc = loc-1: GO
TO 120
 210 IF z$ = "n" OR z$ = "N" THEN GO TO 100
 220 IF z$ = "z" OR z$ = "Z" THEN LET z$ = "00"
 230 LET ans = (CODE z$(1)-48)*16
 240 IF ans>9*16 THEN LET ans = ans-7*16
 250 LET L = CODE z$(2)-48: IF L>9 THEN LET L = L-7
 260 LET ans = ans + L
 270 POKE loc,ans
 280 LEC loc = loc + 1
 290 GO TO 120
 800 STOP
1400 REM **subrotina HEX/DEC**
1410 LET x = INT (n/16)
1420 LET y = ((n/16)—INT (n/16))*16
1430 IF x>9 THEN LET x = x + 7
1440 IF y>9 THEN LET y = y + 7
1450 LET x = x + 48: LET y = y + 48
1460 PRINT CHR$ x;CHR$ y;
1470 RETURN
1600 REM **ENTRADA SUB. HEX/DEC**
1605 INPUT "Numero a converter"; n
1607 LET a = n
1610 LET x = INT (n/16)
1620 LET y = ((n/16) - INT (n/16))*16
```

```
1630 IF x>9 THEN LET x=x+7
1640 IF y>9 THEN LET y=y+7
1650 LET x=x+48: LET y=y+48
1660 PRINT a;" = "; CHR$ x;CHR$ y
1670 RETURN
```

Vejamos agora como é usado o programa monitor que acabamos de apresentar. É mais curto do que os programas comerciais do mesmo tipo, e é certamente o mais fácil de utilizar. O primeiro aspecto a notar é que a linha 5 é usada para especificar a RAMTOP recorrendo à instrução CLEAR. É portanto necessário redefinir este valor, dependendo da parte da memória a usar a rotina em código-máquina. Em seguida execute o programa com RUN. Ser-lhe-á pedida a posição inicial — indique-a. Este endereço inicial será impresso no visor, seguido pelo respectivo conteúdo em hexadecimal. Pode em seguida indicar um novo valor, em hexadecimal, não se esquecendo de usar apenas maiúsculas. Alternativamente, o leitor poderá usar uma das seguintes ordens:

P — Copiar o visor para a impressora.
S — Manter o valor numa dada posição e saltar para a seguinte.
V — Converter um número decimal em hexadecimal.
J — Saltar uma posição para trás.
N — Partir de uma nova posição.
Z — Passar a zero o conteúdo da posição.

Depois de indicar o valor, ou executar as ordens P,V ou Z, será impressa no visor a posição seguinte e o seu conteúdo. Para terminar este processo, escreva "fim" quando lhe for pedido um valor. Se bem que este programa seja curto, e relativamente simples, será muito valioso na ausência de um programa Assembler ou de um programa comercial para montagem de código.

Uso de um assembler

O Assembler pode ser carregado no seu computador da forma habitual. Se possuir as versões de 16 e 48K, não se esqueça de carregar a versão apropriada à sua máquina. É importante que

não carregue a versão incorrecta. O Assembler usado na escrita deste livro é chamado Ultra Violet e vendido pela ACS Software.

Se decidir usar o mesmo Assembler beneficiará provavelmente com algumas explicações complementares.

Para carregar este programa utilize LOAD "" (a versão de 16 K possui três partes). Para preparar o Assembler para ser usado carregue em qualquer tecla. Isto bastará para limpar a memória do computador "abaixo" da posição onde se encontra o Assembler. O programa Assembler encontra-se num endereço bastante elevado, podendo ser chamado a partir da Basic.

Depois de feito isto o leitor encontrar-se-á em comando normal Basic. Pode-se introduzir código-máquina num programa quase como se fosse Basic. A única diferença consistirá em que deve existir uma declaração REM antes do próprio código-máquina. A linha que contém a ordem em código-máquina terá a seguinte aparência: primeiro o número de linha, depois uma declaração REM, finalmente as mnemónicas da linguagem Assembly.

A primeira coisa a fazer quando se usa este Assembler consiste em decidir exactamente onde se deseja colocar o código depois de ter sido "assemblado". Existem aqui duas alternativas: a primeira e provavelmente a mais útil consiste em colocá-lo numa posição qualquer da memória acima do programa Basic e abaixo do Assembler.

Nota: O próprio Assembler utiliza, na versão de 48 K, endereços a partir do 60 000, e na versão 16 K a partir de 27 500. Não é portanto possível construir um código-máquina nestas posições, se bem que seja possível rodear o problema como veremos mais adiante.

A segunda alternativa consiste em introduzir o código-máquina numa declaração REM no início do programa Basic. Esta declaração é colocada no início do programa Basic por uma simples questão de conveniência. O endereço habitual para o primeiro caracter depois da declaração REM na primeira linha é 23760. Se resolvermos optar por esta solução, o que talvez não seja aconselhável por enquanto, e é desnecessariamente complicado no Spectrum, devemos escrever um número suficiente de espaços à frente da declaração REM para substituição pelo código-máquina. Se por exemplo se quiser incluir um código de 25 bytes numa declaração REM, é necessário deixar 25 espaços livres para o efeito ao introduzir a linha.

O Assembler reconhece todas as mnemónicas Z80 em letras minúsculas, exactamente tal como aparecem no fim do manual Basic do ZX Spectrum, fornecido juntamente com o seu computador. Existem poucas excepções a esta regra, que não são particularmente importantes e estudá-las-emos mais adiante. Um ponto importante a notar, no entanto, é que todos os números devem ser indicados em decimal, se bem que sejam listados em hexadecimal depois de "assemblados".

Como introduzir o seu programa

A primeira instrução a ser passada para código-máquina é a GO. Não se trata de uma mnemónica Z80, servindo apenas para informar o Assembler sobre o ponto onde reside o programa a "assemblar". A linha seguinte deve conter o endereço concreto a partir do qual deve ser "assemblado" o nosso código. Usa-se para tal a ordem ORG seguida desse endereço. ORG também não é uma mnemónica Z80, servindo para dizer ao "Assembler" onde deve colocar o código-máquina. ORG significa "ORiGem". Se esquecermos qualquer destas duas ordens, o Assembler devolve uma mensagem de erro. Podemos agora começar a indicar as mnemónicas. Pode-se colocar mais do que uma mnemónica em cada linha desde que as separemos por pontos e vírgulas, se bem que seja provavelmente mais fácil e óbvio colocar apenas uma instrução em cada linha. Deixámos portanto espaço no início do programa Basic para inclusão do código-máquina (se quisermos evidentemente colocá-lo numa declaração REM). Dizemos ao computador, através de uma instrução GO, onde se encontra o programa a converter, e dizemos-lhe também, através da ordem ORG, onde desejamos que ele coloque o código já convertido. Podemos agora dar entrada ao programa em linguagem Assembly. Muitas vezes, ao longo de um programa, o leitor desejará incluir alguma observação ou explicação do que está a fazer. Isto pode ser feito ao usar o Assembler, incluindo um ponto de exclamação depois da REM e escrevendo o comentário. O ponto de exclamação é usado para dizer ao Assembler que os códigos ou a palavra que se seguem não devem ser traduzidos, apenas tendo interesse para o programador. Se o leitor esquecer este ponto de exclamação o Assembler tentará converter o que se encontra a seguir, provocando provavelmente o aparecimento de uma mensagem de erro.

O leitor pode em seguida escrever o programa a converter. Deve fazê-lo em minúsculas, pois o computador compreenderá o que deseja.

Nota: Pode-se usar maiúsculas dentro de um comentário, mas só minúsculas nas instruções a converter.

Depois de ter escrito todas as mnemónicas deve terminar com a palavra Fim (Finish), que servirá para indicar à máquina que chegou ao fim da rotina que deseja converter. Finish não é também uma mnemónica Z80.

O leitor disporá agora de um programa em linguagem Assembly, pronto para ser convertido em linguagem-máquina. Antes de o fazer, porém, é útil verificar se não cometeu quaisquer erros na escrita do programa. Obviamente, se houver erros, é fácil alterar a declaração errada, o que é feito do mesmo modo que nas listagens Basic. Para dizer ao Assembler que pretende converter as mnemónicas já escritas, escreva RANDOMIZE SPACE USR 60000 se tiver um Spectrum de 48 K, ou RANDOMIZE SPACE USR 27500, se tiver um Spectrum de 16 K.

As mnemónicas surgirão em seguida no visor juntamente com os respectivos códigos em linguagem-máquina. Se houver algum erro, por exemplo declarações que o Assembler não pode compreender, será impresso um código de erro e ser-lhe-á pedido para rectificar o problema. Note por favor que um Assembler não verificará o programa. Um ponto interessante a notar é que quando o programa está a ser convertido a listagem do código é impressa duas vezes. Isto deve-se ao facto de o Assembler fazer duas passagens pelo programa antes de guardar a versão final na memória. As razões disto serão apresentadas mais tarde.

Sabemos agora o que acontece em teoria; vejamos portanto como podemos passar à prática. A primeira coisa a recordar é que desejamos pôr o nosso código-máquina, não numa declaração REM mas numa dada posição de memória, digamos a 30000. O programa que vamos converter é apresentado abaixo, servindo para rolar pixels para a direita.

```
ld hl,22527
push hl
ld de,23328
ld bc,32
lddr
pop hl
```

HL = Endereço inferior do bloco inferior
B = Nº de blocos

```
        ld b,3
A       push bc
        ld b,8
B       push bc
        push hl
        ld b,8
C       push bc
        push hl
        push hl
        pop de
        dec h
        ld bc,32
        lddr
        pop hl
        dec h
        pop bc
        djnz ,C
        inc h
        ld de,1760
        push hl
        add hl,de
        pop de
        ld bc,32
        lddr
        pop hl
        ld de,32
        sbc hl,de
        pop bc

        djnz,B
        ld de,1792
        sbc hl,de
        push hl
        ld de,32
        add hl,de
        push hl
        pop de
        pop hl
        push hl
        ld bc,32
        lddr
        pop hl
        pop bc
        djnz,A
        ld de,32
        add hl,de
        ex de,hl
        ld hl,23328
        ld b,32

D       ld a,(hl)
        nop
ld a,0 para não reintroduzir pixels
        ld (de),a
        dec de
        dec hl
        djnz ,D
        ret
```


O programa anterior é apresentado em linguagem Assembly, servindo o Assembler para convertê-lo em linguagem-máquina. A primeira operação consiste em introduzir tudo isto no Assembler. Teremos antes do mais de carregar o Assembler, o que é feito escrevendo LOAD "" e verificando se carregamos o programa correcto (48 ou 16 K). Em qualquer dos casos o programa demorará menos de um minuto a carregar. Se estiver a usar o Assembler Ultra Violet, ser-lhe-á apresentado um visor grande, que desaparecerá assim que carregar em qualquer tecla, e observará então a mensagem normal da máquina quando é ligada, ou seja, © 1982 Sinclair Research Ltd. Está agora pronto a escrever as instruções em linguagem Assembly. Ser-lhe-á tão fácil fazê-lo como escrever um programa em Basic. É muito importante recordar que todas as instruções em linguagem Assembly, e as usadas para comandar o programa Assembler, devem ser introduzidas numa linha depois de uma declaração REM. Isto permite que as instruções sejam apenas compreendidas pelo Assembler, e não pela Basic.

É necessário dizer ao Assembler que existe um programa em linguagem Assembly para ser convertido. Isto é feito inserindo a

instrução "GO" na primeira linha do programa. Por exemplo: 5 REM go. Em seguida devemos especificar o local onde queremos guardar a rotina em código-máquina. Isto é feito recorrendo à instrução ORG, seguida do endereço onde desejamos colocar o primeiro byte do código convertido; o resto do código será colocado sequencialmente a partir deste endereço. Assim, por exemplo, se a sua rotina tiver 10 bytes de comprimento, e pedir ao computador que a guarde na posição 28000, a rotina ocupará todos os bytes entre 28000 e 28009 (num total de dez bytes). Como desejamos colocar o programa anterior a partir da 30000, a linha de Basic seguinte deve ser: 8 REM org 30000.

Estamos praticamente prontos a escrever as instruções em mnemónicas, mas antes de o fazermos é útil inserir um comentário no início do programa a fim de tornar possível reconhecê-lo mais tarde. O modo de incluir um comentário num programa consiste em inserir um ponto de exclamação a seguir ao número de linha e à declaração REM, escrevendo depois o comentário. No caso do programa anterior seria suficiente escrever que se trata de um programa para rolar os pixels para baixo, bastando escrever: 12 REM ! Rolar pixels — para baixo.

Está tudo preparado para iniciar a escrita das instruções Assembly. Não se esqueça de, tal como faria no caso de um programa Basic, incluir números de linha, nem de incluir uma declaração REM antes de cada instrução. A última, que indicará o fim da rotina, será "Finish". Esta ordem deverá ser escrita do mesmo modo que anteriormente: Número de linha, Rem, finish. A conversão do programa pode iniciar-se em seguida.

Se por qualquer razão tiver problemas na escrita do programa, e se quiser verificar o que escreveu, pode comparar com a listagem que apresentamos a seguir. Não se preocupe com os números das linhas, mas verifique se as instruções são iguais e se encontram escritas pela mesma ordem.

```
  1 REM go
  2 REM org 30000
 10 REM ld hl,22527
 20 REM push hl
 30 REM ld de,23328
 40 REM ld bc,32; lddr; pop hl;! HL=BOT. END.
INFERIOR DO FLOCO INFERIOR
```



```
 50 REM ! B=No. de blocos; ld b,3; A; push bc; ld b,8; B;
push bc; pusch hl; ld b,B; C; push bc; push hl; push hl; pop
de; dec h; ld bc,32; lddr; pop hl; dec h; pop bc; djnz,C; inc h;
ld de,1760; push hl; add hl, de; pop de; ld bc,32; lddr; pop hl;
ld de,32; sbc hl, de; pop bc; djnz, B; ld de,1792; sbc hl, de;
push hl; ld de,32; add hl, de; push hl; pop de; pop hl; push
hl; ld bc,32; lddr; pop hl; pop bc; djnz, A
 60 REM ld de,32; add hl, de; ex de, hl; ld hl, 23328; ld
b,32; D; ld a, (hl); nop; ! Ld a,0 para não reintroduzir pixels;
ld (de), a; dec de; dec hl; djnz, D; ret 1000 REM finish
```

A conversão é realizada acedendo a uma rotina em código previamente carregada no computador. Como existem duas versões deste programa, existem dois endereços diferentes para guardar esta rotina. Se possuir um Spectrum de 16 K deve escrever a ordem seguinte:

RANDOMIZE USR 27500

Se possuir um Spectrum de 48 K, deve executar a ordem:

RANDOMIZE USR 60000

Importante: não se esqueça de que as instruções dadas acima para a conversão, e também o modo como introduzimos a rotina em linguagem Assembly, só são aplicáveis usando o Assembler ACS. Existem actualmente vários programas Assembler no mercado, e é provável que haja ainda mais quando este livro tiver sido publicado; não posso evidentemente definir aqui o modo como funcionam dado que podem existir grandes diferenças. Preferi concentrar-me no uso de um dado Assembler porque penso que é o mais fácil de usar. Se o leitor já possui um Assembler, ou decidir comprar um diferente daquele que uso aqui, necessitará de estudar convenientemente o seu manual de instruções. Isto significa que a última listagem que apresentámos não será provavelmente utilizável. Como é óbvio, no entanto, a rotina listada um pouco antes funcionará do mesmo modo qualquer que seja o Assembler utilizado.

Voltemos à nossa tarefa. Temos um programa Basic pronto, e sabemos como instruir o programa Assembler de modo a con-

verter as instruções que introduzimos. Vamos portanto continuar. Depois de executar a ordem de conversão dos códigos, surgirá no visor uma imagem multi-colorida. A conversão ainda não está terminada. Se carregar na tecla "P", o computador enviará uma listagem da sua rotina em código-máquina para a impressora. Carregue na barra de espaços se quiser abortar a conversão — carregar em qualquer outra tecla provocará a continuação da conversão. É essencial recordar que só pode usar a ordem "P" para imprimir na impressora se a rotina em código-máquina for apresentada pela segunda vez, ou na sua segunda "passagem". Tendo carregado na tecla Enter, e se tiver surgido no visor uma mensagem "No Error" (Ausência de erros), o leitor poderá mandar executar o código-máquina a partir do endereço 30 000 do modo habitual (RANDOMIZE USR 30 000 ou PRINT USR 30 000, etc.)

Um erro no seu programa provocará uma das seguintes consequências:

Se Go, Finish ou Org estiverem incorrectas, ocorrerá um erro antes de se iniciar a conversão.

Se for detectado um erro durante a conversão, o Assembler parará imprimindo uma ou duas mensagens de erro. A primeira é uma mensagem de erro que indica o número da linha, o número da declaração e o tipo de instrução onde ocorreu o erro. Isto facilita ao leitor voltar ao seu programa Basic e descobrir onde se encontra o erro. A segunda mensagem é uma das mensagens Sinclair normais. Há então três possibilidades. Se tiver indicado um número errado, ou seja, se tentar indicar um número superior a 255 no caso de um registo de um só byte ou de 65535 no de um registo de dois bytes, o Assembler diminuirá repetidamente 256 ou 65536 do número original até obter um valor que possa ser usado. Infelizmente, isto não acontecerá se o leitor tiver indicado um número excessivo para o deslocamento num salto relativo. Se bem que ainda não tenhamos falado destes saltos relativos, o que acontecerá dentro em breve, convém desde já ter este aspecto em conta. No caso de não encontrar um resultado apropriado, o programa produzirá a mensagem "B Integer Out of Range" (Inteiro fora da gama aceite).

Existem ainda outros casos em que é possível detectar erros, mas estes não são importantes por agora.

Agora que temos uma rotina em código-máquina na memória, que deve ser imediatamente passada para uma cassette para o ca-

so de ocorrer alguma falha durante a sua execução, podemos simplesmente recarregar o programa. A gravação de uma rotina em código-máquina está bem documentada no próprio manual Spectrum, mas vou recordar rapidamente os pontos essenciais.

É apenas necessário escrever SAVE "Nome" CODE 30 000,100. Isto diz muito simplesmente ao computador para passar para a cassette o programa em código-máquina (sabe que se trata de código-máquina porque o leitor escreveu a palavra-chave CODE seguida de dois números) que se encontra nos endereços 30 000 a 30 000 + 100, sob o nome escrito entre aspas. Por exemplo, se desejamos gravar a rotina em código indicada acima, com o nome pixel-d, escrevemos o seguinte: SAVE "pixel-d" CODE 30 000,100. Não se esqueça de que está apenas a gravar o programa em código-máquina, e não o Assembler ou o programa Basic original. Voltar a carregar este programa é ainda mais fácil, bastando neste caso escrever LOAD "pixel-d" CODE. Esta ordem carregará o programa em código-máquina que possui o nome "pixel-d" para o endereço do qual foi gravado. Quando a operação estiver terminada surgirá na parte inferior do visor uma mensagem indicando o facto.

Disassemblers

Muitas vezes é útil fazer exactamente o contrário do que estivemos a expor. Por outras palavras, em vez de convertermos uma lista de mnemónicas em código-máquina, podemos querer converter o código-máquina numa lista de mnemónicas. É isto que acontece muitas vezes quando se encontra na memória da máquina uma rotina que não foi escrita por nós, ou que nós próprios escrevemos mas que queremos verificar. Uma outra vantagem da maior parte dos Disassemblers é que imprimem no visor não só as mnemónicas mas também os códigos hexadecimais correspondentes às instruções. É possível usar esta informação de diversos modos, alguns dos quais explicaremos mais adiante. Já utilizei o Disassembler para produzir listagens de programas apresentados neste livro. Usar um Disassembler é muito mais fácil do que usar um Assembler, e nestas condições vou perder pouco tempo com o assunto.

Carregar o Disassembler da "ACS Software" é semelhante a carregar o Assembler — e não convém uma vez mais esquecer de

verificar se a versão usada corresponde ao seu computador. É igualmente importante recordar que é possível ter simultaneamente em memória o Assembler e o Disassembler em ambas as versões. Dispõe-se assim de um instrumento de programação bastante potente mesmo que se possua apenas a versão do Spectrum com 16 K. É porém necessário carregar o Assembler antes do Disassembler. Depois de ambos terem sido carregados no computador é possível executar o Disassembler recorrendo a uma das seguintes ordens:

Spectrum 48 K: RANDOMIZE USR 54 000
Spectrum 16 K: RANDOMIZE USR 26 600

Quando é dada uma destas ordens surge na parte superior do visor a pergunta "Endereço inicial?". É fácil indicar então, em decimal, o endereço a partir do qual se pretende fazer a conversão para menmónicas. Se cometer um erro ao escrever o endereço não use "Delete"; em vez disso, carregue na tecla "E". Assim que tiver indicado o número correcto é apresentada no visor a primeira página de código convertido. Se quiser continuar, basta escrever "C"; se quiser voltar à pergunta "Endereço inicial?" escreva "R". Se quiser fazer uma cópia do visor actual para a impressora escreva "P", e se quiser abandonar o Disassembler e voltar à Basic, escreve "E".

Uma das características mais úteis tanto do Assembler como do Disassembler é que se mantêm em memória enquanto estamos a usar o nosso próprio programa Basic ou em código-máquina, desde que o programa em código-máquina não esteja obviamente colocado nos mesmo endereços que o Assembler ou Disassembler e não seja afectado por uma declaração NEW. É certo que a declaração NEW é dada como apagando completamente a memória; sim, mas fá-lo apenas até à RAMTOP, como se disse no início do livro. Esta limitação serve precisamente para permitir que coisas como os gráficos definidos pelo utilizador, ou programas ou dados em bytes possam ser resguardados; não podem portanto ser corrompidos pelo programa Basic em funcionamento nem por uma ordem NEW.

Como definimos a posição da RAMTOP? Usamos simplesmente a ordem CLEAR xxxxx (onde xxxxx indica a RAMTOP, ou maior endereço ainda utilizável em Basic). Isto significa que toda a memória depois deste endereço estará protegido. Por

exemplo, para proteger toda a memória a partir de 28 000, permitindo que um programa de remuneração, o Assembler e o Disassembler se mantenham em memória, escreveríamos CLEAR 27 999. Recordando que o endereço indicado faz parte ainda da área Basic, é necessário indicar como RAMTOP o endereço imediatamente inferior ao primeiro que desejamos proteger.

O leitor verificará que muitos programas incluem a instrução CLEAR na listagem Basic, garantindo a protecção adequada. É igualmente interessante notar que se sairmos do Disassembler, veremos que no curto programa Basic já existente em memória existe uma declaração CLEAR. Recorde ainda que CLEAR limpa o visor, mas não limpa nenhuma porção da memória acima da RAMTOP.

Interrupção de código-máquina

O modo como a tecla BREAK funciona em Basic é muito simples; o computador lê simplesmente o teclado, e se a tecla em causa se encontra carregada a máquina limita-se a saltar para fora do que está a fazer e imprime uma mensagem de erro, voltando ao comando Basic. Infelizmente não podemos fazer isto directamente em código-máquina, porque esta possibilidade não existe em hardware. Podemos porém rodear o problema usando uma curta rotina BREAK feita por nós, que pode ser guardada juntamente com outras rotinas em código-máquina e que pode ser acedida quando quisermos para verificar se o utilizador actua sobre a tecla BREAK. A curta rotina em código-máquina apresentada a seguir faz precisamente isto. Nem todas as instruções usadas no programa foram discutidas até agora, e de facto algumas delas, em particular a instrução RRA, não serão mencionadas. Basta no entanto saber que no caso de a tecla BREAK ser carregada este programa provocará imediatamente um retorno à Basic.

```
ld a,127
in a,(254)
rra
ret nc
```

Por agora só nos é possível incluir esta rotina como parte do nosso programa, mas mais adiante aprenderemos a usá-la como

subrotina da rotina principal. Trata-se no entanto de um instrumento muito útil e potente. Mais adiante falarei ainda sobre o modo de usar esta rotina não só para detectar se a tecla BREAK foi carregada, como ainda para verificar se se carregou em qualquer outra tecla. Continuemos.

Monitores

O leitor já perguntou certamente a si próprio o que poderá fazer se não dispuser de um Assembler, ou não quiser comprar um. Não existe nenhuma forma simples de obter as vantagens de um Assembler sem o comprar, mas continua a ser possível dar entrada a código na memória, se bem que directamente em linguagem-máquina, e dispondo ainda de algumas das vantagens oferecidas por um Assembler — usamos então um Programa Monitor. Este possui ainda algumas vantagens que não são inerentes aos Assemblers e que contrabalançam um pouco a necessidade de converter as mnemónicas em código-máquina para utilização das tabelas existentes no final do livro.

Programas de determinação de erros (Debugging)

O leitor verificará que mesmo no caso de ter revisto cuidadosamente uma dada rotina em código-máquina, é provável que esta contenha erros. Apesar de tudo o que faça, os erros parecem surgir sozinhos... Não é possível recuperar de um "crash" (estoiro) da máquina, sendo portanto particularmente importante que descubra os erros de um programa antes de o pôr em execução. Por esta razão, e por outras que se tornarão evidentes, é muito útil possuir um programa Debugger. Estes programas são vendidos do mesmo modo que os Assemblers ou Disassemblers. Para informação do leitor, o programa Debugger que utilizei neste livro e será usado como exemplo não foi comercializado pela mesma casa que vende o Assembler o Disassembler, se bem que esta deva ter um à venda quando este livro for publicado. Foi usado um da Artic Computing.

Carregar no Spectrum este programa Debugger é extremamente simples, bastando escrever LOAD "", mas não esqueça uma vez mais de carregar a versão correcta para a máquina que usa. Depois de o programa estar carregado, o leitor terá várias

opções; mas como o programa é concebido para uso com as suas próprias rotinas em código-máquina parece lógico carregar primeiro uma destas, por exemplo a rotina para rolar pixels que apresentámos no início deste capítulo. Se bem que não existam erros neste programa, pode apesar de tudo constituir um bom exemplo. Por outro lado, como não é possível carregar códigos que não tenham sido passados para a cassette usando este mesmo programa Debugger, é necessário sair primeiro deste, voltando à Basic, antes de carregar aquela rotina. Isto é feito carregando simplesmente na tecla "X" e em Enter. Depois de ter feito isto, pode carregar normalmente a rotina em causa. Em seguida volta a accionar o programa Dedugger escrevendo PRINT USR 30 884 no caso de um Spectrum de 16 K, ou PRINT USR 63652 no caso da máquina de 48 K. É muito importante empregar os números correctos (encontram-se indicados nas instruções que acompanham o programa). Note que todas as funções se encontram listadas nas instruções deste programa, mas só são citadas aqui como referência; não pretendo explicar todas as ordens e opções disponíveis neste programa Debugger. Vejamos no entanto as mais importantes:

"Z" permite "disassemblar" (converter para mnemónicas) uma pequena área de RAM — tal como faz o Disassembler que referimos anteriormente, se bem que não de um modo tão sofisticado. Para converter a área onde reside a rotina de movimento dos pixels, escreva simplesmente: Z 7530. Se por qualquer razão quiser parar a conversão, carregue em BREAK. Não se esqueça de carregar em Enter sempre que escrever as instruções do programa Debugger.

Nota: Todos os valores usados neste programa Debugger são indicados em hexadecimal, o que significa que utilizam um máximo de quatro algarismos, não sendo permitidos números decimais. Por esta razão pode ser útil recordar capítulos anteriores e rever rapidamente o modo como se fazem conversões entre decimal e hexadecimal. Para auxiliar o leitor, no entanto, inclui-se uma lista de todos os valores binários, decimais e hexadecimais entre 0 e 255 no Apêndice A, que lhe pode ser bastante útil ao realizar a conversão. É igualmente possível executar uma rotina em código-máquina do próprio programa Debugger, recorrendo à ordem G seguida do endereço em hexadecimal onde se inicia a rotina, seguido de Enter. Pode-se então observar o conteúdo dos

registos principais carregando em "D". Surgirão assim na parte superior do visor os valores dos principais registos.

Esta possibilidade é muito útil no final de um programa, mas não seria igualmente útil descobrir o conteúdo dos principais registos a meio do programa? Isto pode ser conseguido muito facilmente. Quando o programa atinge certos endereços volta automaticamente para o programa Debugger. Pode-se usar este para listar os registos, e como o leitor descobrirá mais tarde é até possível listar as flags. Para definir um desses pontos de paragem da conversão escreve-se simplesmente "Q", seguido do endereço em hexadecimal do ponto de paragem. Em seguida executa-se a rotina em código-máquina escrevendo "G", como já se disse. Assim que o programa cumpre a paragem, o ponto de paragem é anulado.

Experimente em seguida "disassemblar" o programa, decidir onde deve colocar os pontos de paragem, inseri-los, executar o programa, observar o conteúdo dos registos. Usando a ordem "F" seguida de Enter, pode também imprimir o conteúdo das flags. Existem algumas outras ordens menos espectaculares no programa Debugger da Artic que convirá mencionar rapidamente. É possível:

1. Enviar uma mensagem para uma certa parte da memória, escrevendo simplesmente a mensagem em causa no teclado. A máquina converte-a no valor hexadecimal apropriado.
2. Procurar um determinado valor num bloco de memória, sendo impresso o endereço onde se encontra.
3. Copiar um bloco de memória para outro local.
4. Imprimir o conjunto alternado de registos, que ainda não estudámos.
5. Ordenar ao programa que substitua um determinado valor em todos os endereços onde se encontra por outro.
6. Levar o programa Debugger a carregar e passar para cassette os programas que está a tratar.
7. Modificar o conteúdo de vários bytes de memória, e dar entrada a código-máquina para certos endereços, executando-os em seguida.
8. Dar valores bem definidos a registos específicos, de tal modo que se possam simular certos acontecimentos no interior do programa
9. Imprimir os códigos de caracter da sua rotina, a fim de poder passá-los para uma declaração REM.

Que programa?

Se o leitor só está disposto a adquirir um destes programas, talvez seja melhor começar pelo Debugger. Não só porque lhe será extremamente útil quando se tratar de descobrir os erros de um programa, mas também porque é um óptimo auxiliar para a escrita de programas e ainda lhe permite compreender o modo como funciona um programa e o próprio computador em geral.

Se decidir dedicar-se mais ao código-máquina, e quiser escrever programas mais compridos e complicados, é muito provável que um Assembler seja o instrumento mais útil, desde que seja acompanhado por um Disassembler. Em seguida poderá comprar o programa Debugger.

E finalmente...

Antes de começar a escrever programas, questão que será estudada no capítulo seguinte, passarei em revista quatro instruções muito úteis que aliás já usámos mas que o leitor não terá compreendido totalmente ou que foram referidas mas não explicadas. Tal como tudo o que aprendemos até agora, estas instruções tornar-se-ão muito mais claras quando forem utilizadas. Nestas condições, o capítulo seguinte trata apenas da escrita de programas em código-máquina. Será dada também muita atenção à descrição cuidadosa de cada instrução ou método de trabalho necessário para produzir um bom programa. Iremos para já observar quatro instruções: INC, DEC, NOP, RET.

INC

INC é mais uma das instruções que nos surgem em duas formas. Pode ser usada sobre um registo simples ou sobre um par de registos. Uma das maiores vantagens desta ordem consiste em poder ser usada sobre qualquer registo. INC X provocará o aumento do conteúdo do registo X de uma unidade. Nestas condições, por exemplo, se o registo A contém o valor 5, e for executada a ordem INC A, o valor do registo A passará a 6. O efeito

será exactamente o mesmo quer se trate do registo A ou de qualquer dos registos B, C, D, E, H, L. Se o conteúdo do registo for 255 e for incrementado voltará a zero; mas neste caso não será afectada a flag "carry", ao contrário do que acontece para uma instrução de soma (ADD). Este aspecto é muito importante, devendo ser recordado quando, mais adiante, o leitor quiser verificar o valor de um registo para decidir sobre o que acontecerá no resto do programa.

O diagrama que se segue é uma representação simples do funcionamento desta instrução.

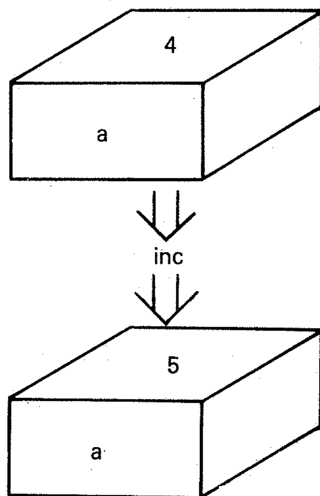


FIG. 15

Se o leitor quiser incrementar o conteúdo de uma variável em Basic, terá de usar a instrução LET B=B+1 no caso de a variável em causa ser designada B. É óbvio, neste exemplo, que o equivalente em código-máquina é consideravelmente mais curto, permitindo portanto poupar memória, e extremamente rápido. As mnemónicas e correspondentes códigos hexadecimais das instruções de incrementação são apresentadas a seguir:

INC A	3C
INC B	04
INC C	0C
INC D	14
INC E	1C
INC H	24
INC L	2C

É possível incrementar o valor de um par de registos, tal como se podem incrementar registos simples. A única diferença reside no facto de a instrução não afectar qualquer flag, incluindo a flag "carry".

As mnemónicas e os códigos hexadecimais destas instruções são apresentadas a seguir:

INC BC	03
INC DE	13
INC HL	23

O funcionamento destas instruções não é muito difícil de compreender, se bem que possa haver alguma dificuldade inicial. Para o caso de o leitor ter algum problema, vamos ilustrar a questão com um pequeno programa em código-máquina. Executando este programa por uma ordem Basic (usando a declaração PRINT USR), a máquina imprimirá o valor final do registo BCx. É interessante jogar um pouco com o conteúdo de BC na primeira declaração, notando o modo como as diversas alterações afectam a resposta.

```
org 30000
30030 01 00 00        ld bc,0000
30003 03              inc bc
30004 C9              ret
```

Existe ainda uma função da ordem INC que não referimos. É possível incrementar o conteúdo de uma posição de memória. Por exemplo, se o leitor carregar primeiramente no par HL o endereço da posição que deseja incrementar, e em seguida executar a instrução INC (HL), o conteúdo do endereço presente em HL será incrementado de uma unidade (note que o conteúdo de HL

não será alterado). O diagrama seguinte ilustra este assunto com maior clareza.

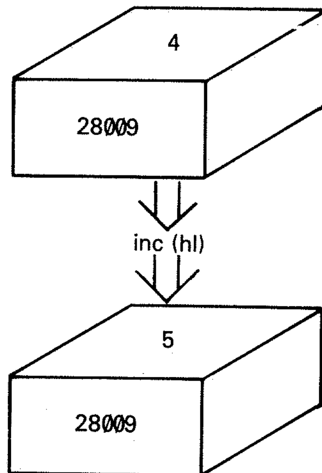


FIG. 16

O código hexadecimal desta instrução é 34. Não se esqueça de que instrução só pode ser aplicada ao registo HL.

Introduza agora na máquina o programa listado em seguida, execute-o e verifique o valor da posição 28012. Execute novamente a rotina e verifique de nodo o conteúdo desse endereço. Da segunda vez terá aumentado de 1. Note que a operação de incrementação do valor de um endereço apenas tem efeitos sobre esse endereço, não sendo portanto possível ultrapassar o valor 255, correspondente ao maior valor que pode encontrar-se em qualquer posição de memória.

Vejamos o programa:

```
org 30000
30000 21 6C 6D          ld hl,28012
30003 23                inc(hl)
30004 C9                ret
```


DEC

Esta instrução é muito semelhante a INC, mas neste caso o valor do registo ou endereço de memória é decrementado de uma unidade em vez de ser aumentado. Por exemplo, se decidirmos decrementar A quando este registo contém o valor 7, o resultado será 7-1, isto é, 6. O diagrama que se segue ilustra claramente o que se passa.

Em seguida apresentamos uma lista das mnemónicas e dos códigos hexadecimais destas instruções:

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

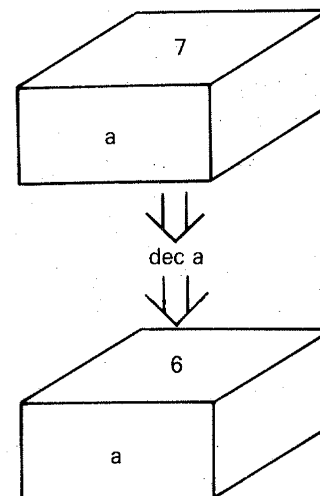


FIG. 17

O leitor já é certamente capaz de converter o programa exemplo usado anteriormente de modo a verificar o funcionamento tanto das ordens INC como DEC. No caso de o assunto ainda lhe parecer confuso, no entanto, vejamos um modo de alterar a rotina:

```
org 30000
30000 01 00 00        ld bc,0000
30003 0B              dec bc
30004 C9              ret
```

Note-se que também neste caso a flag "carry" não é alterada quando a instrução de decrementação é executada tanto num registo simples como num par de registos. As mnemónicas e os códigos hexadecimais deste conjunto de instruções são indicados em seguida:

```
DEC BC        0B
DEC DE        1B
DEC HL        2B
```

Como pode verificar observando as mnemónicas e códigos hexadecimais indicados, existe uma certa relação entre estes e as mnemónicas. Por exemplo, todas as instruções de decrementação sobre pares de registos possuem um B como segundo algarismo. É útil recordar pequenos pormenores como este — torna-lhe a vida muito mais fácil se não dispuser de um Assembler. Faça um intervalo de alguns minutos para consultar outras páginas deste livro tentando encontrar outras relações deste tipo.

Tal como acontece no caso da instrução de incrementação, é possível decrementar o conteúdo de uma dada posição de memória quando o endereço desta se encontra no par de registos HL. Encontramos em seguida um exemplo disto

```
org 30000
30000 21 6C 6D        ld hl,28012
30003 2B              dec(hl)
30004 C9              ret
```

Depois de executar esta rotina, verifique o valor da posição 28012 usando a declaração PEEK, e em seguida volte a executar

a rotina e a determinar o conteúdo desse endereço. Em todos os casos, excepto quando o valor inicial é zero, o conteúdo terá diminuído de 1. Se, por outro lado, o conteúdo da posição em causa já é 0, passará a 255 ao ser decrementado.

RET

Neste momento o leitor já compreendeu certamente que esta instrução serve para indicar à máquina o final de uma rotina, reenviando o comando para a Basic. A instrução RET é sempre necessária no final de um programa, a menos que o leitor use uma das variantes que serão indicadas mais tarde. Estas variantes não afectam o que estamos agora a dizer. O código hexadecimal da mnemónica "RET" é muito fácil de recordar, e dentro de poucas páginas já se terá transformado numa espécie de segunda natureza do leitor — é C9. A razão desta familiaridade é que o leitor estará sempre a usá-la...

Esta ordem é muito fácil de compreender — não serve absolutamente para nada... Quando o computador encontra esta instrução, cujo código é 0, não faz nada, limitando-se a passar à instrução seguinte e deixando exactamente iguais os conteúdos de todos os registos, endereços de memória e flags. Pode parecer estranho que alguém possa necessitar de uma instrução como esta, mas de facto trata-se de uma ordem extremamente útil. Muitas vezes é necessário deixar no interior de uma rotina certas áreas sem nada, permitindo por exemplo a inclusão de novas ordens mais tarde. Isto é semelhante ao que acontece quando se deixam números de linha livres entre instruções Basic. Por outras palavras, os programas em Basic nunca contêm linhas seguidas, por exemplo 1, 2, 3, 4, etc., porque não deixam possibilidades de manobra ao programador. Do mesmo modo, pode ser muito útil dispor de uma ordem que assegure, quando necessário, a não execução por parte do computador.

Até agora concentrámo-nos essencialmente no lado teórico do código-máquina, se bem que eu tenha usado exemplos e incluído alguns programas. No entanto, o principal interesse do código-máquina é a sua utilização. O capítulo que se segue tratará portanto do modo de construir a "ideia" de um programa, codificá-la, e finalmente verificar esta codificação e executá-la. Antes de abandonarmos este capítulo, no entanto, vejamos um programa

complementar do já apresentado antes. Permite rolar pixels para cima. Utilize um Assembler, ou um Monitor, para introduzir este programa na máquina.

```
org 30000
30000 21 00 40        ld hl,16384
30003 E5              push hl
30004 11 00 5B        ld de,23296
30007 01 20 00        ld bc,32
30010 ED B0           ldir
30012 E1              pop hl
```

HL = Endereço superior do bloco superior

B = Nº de blocos

```
30013 06 03           ld b,3
A
30015 C5              push bc
30016 E5              push hl
30017 06 08           ld b,8
B
30019 C5              push bc
30020 E5              push hl
30021 06 07           ld b,7
C
30023 C5              push bc

30024 E5              push hl
30025 E5              push hl
30026 D1              pop de
30027 24              inc h
30028 01 20 00        ld bc,32
30031 ED B0           ldir
30033 E1              pop hl
30034 24              inc h
30035 C1              pop bc
30036 10 F1           djnz,C
30038 E5              push hl
30039 11 E0 06        ld de,1760
30042 ED 52           sbc hl,de
30044 D1              pop de
30045 01 20 00        ld bc ,32
30048 ED B0           ldir
30050 E1              pop hl
30051 11 20 00        ld de,32
30054 19              add hl,de
30055 C1              pop bc

30056 10 D9           djnz,B
30058 11 E0 06        ld de,1760
30061 19              add hl,de
30062 E5              push hl
30063 11 20 00        ld de,32
30066 19              add hl,de
30067 D1              pop de
30068 01 20 00        ld bc,32
30071 ED B0           ldir
30073 E1              pop hl
30074 11 00 08        ld de,2048
30077 19              add hl,de
30078 C1              pop bc
30079 10 BE           djnz,A
30081 11 20 00        ld de,32
30084 ED 52           sbc hl,de
30086 EB              ex de,hl
30087 21 00 5B        ld hl,23296
30090 06 20           ld b,32
D
30092 7E              ld a,(hl)
30093 00              nop
```

LD A,0 Para não reintroduzir os pixels

```
30094 12              ld (de),a
30095 13              inc de
30096 23              inc hl
30097 10 F9           djnz,D
30099 C9              ret
```

SABE COMO UTILIZÁ-LO?

VII

COMO ESCREVER UM PROGRAMA

O leitor já está bastante avançado no que se refere a aprender o modo de usar o código-máquina e as instruções que este contém. É chegado o momento de começar a construir os seus próprios programas nesta linguagem. Usando apenas algumas ordens o leitor é já capaz de:

1. Somar dois registos
2. Somar valores a registos
3. Subtrair um registo de outro
4. Subtrair números dos registos
5. Incrementar um registo
6. Incrementar o valor de uma posição de memória
7. Decrementar um registo
8. Decrementar o valor de uma posição de memória
9. Carregar um valor num registo
10. Carregar num registo o valor que se encontra noutro
11. Carregar num registo o conteúdo de uma dada posição de memória
12. Carregar numa posição de memória o conteúdo de um dado registo
13. Voltar ao comando Basic e não fazer nada...

Com estes conhecimentos, podemos construir já rotinas em código-máquina bastante interessantes. No entanto, é chegado também o momento de o leitor descobrir que não é tão fácil escrever um programa em código-máquina como o é em Basic — mas se perseverar vencerá facilmente este problema. Em vez de se sentar à frente do teclado e começar a escrever, como faria em

Basic, tem de abordar a construção do programa de uma forma lógica e sistemática; deve decidir fase a fase o que deseja fazer e o melhor modo de o fazer. Verifique sempre se não cometeu quaisquer erros dado que não é possível recuperar de um "crash" em código-máquina. Um outro factor importante é que o programa em código-máquina não é tão fácil de alterar como um programa Basic. Nestas condições, é muito útil dispor de uma boa documentação sob a forma de fluxogramas e notas àcerca do modo exacto como se escreveu o programa. Em seguida apresento um esboço da construção dos meus programas. O leitor acabará por definir um modo seu de construir programas, mas por agora talvez tire algum benefício do estudo dos meus:

1. *Ideia geral*

Decida precisamente o que quer fazer com o programa ou rotina. Isto envolverá observar cuidadosamente o problema que quer resolver ou as ideias que quer aplicar, e decidir quais os resultados pretendidos. Escreva tudo isto e utilize o que escreveu como referência durante a construção do programa.

2. *O fluxograma*

O fluxograma garante a divisão da ideia geral nas suas partes componentes. A construção de cada fase torna-se assim muito mais simples. Por agora não é necessário entrar em pormenores sobre o que se pretende conseguir em cada fase, mas apenas definir a ordem pela qual se deseja executar as várias partes que constituem a ideia geral.

3. *Desenvolvimento da estrutura*

Isto obriga a pensar em todo o programa, fase a fase, analisando qual o movimento ou operação que se deseja ver realizada pelo computador em cada momento. Deve-se manter presente a gama de ordens à nossa disposição. Trabalhando de um modo organizado torna-se fácil aceder à fase seguinte.

4. *O fluxograma final*

Este é basicamente uma amálgama de cada uma das fases par-

ticulares envolvidas no desenvolvimento da rotina, apresentadas sequencialmente e logicamente de uma forma fácil de compreender.

5. *Conversão*

Isto não significa transformar linguagem assembly em linguagem-máquina, mas sim a operação Basic de transformação daquilo que se escreveu sob a forma de declarações curtas em instruções de linguagem assembly. Estas podem depois ser transformadas em mnemónicas, isto é, numa versão compreensível pelo Assembler. As instruções são então ordenadas de uma forma lógica, do modo que se pretende vê-las executadas pelo computador.

6. *Execução "a seco"*

Passe agora o programa passo a passo, não usando o computador, mas papel e caneta — registando o conteúdo dos registos usados, o conteúdo de quaisquer endereços específicos que possam ser alterados, e o estado das próprias "flags". Se tudo correr bem está pronto a utilizar o Assembler, mas se tal não acontecer terá pelo menos evitado todo o trabalho de conversão, introdução na máquina e execução de um programa com erros.

7. *Introdução do programa na máquina*

É muito importante conhecer bem o modo de funcionamento do seu Assembler, sendo por isso que decidimos utilizar apenas um ao longo de todo este livro.

8. *Verificação*

Antes de prosseguir deve-se verificar, através do uso de um Disassembler ou por outro tipo de exame, se se cometeu algum erro na introdução do programa.

Muitas vezes o Assembler detecta erros, mas é também vulgar que este programa não os detecte e faça a conversão de uma instrução errada. Ao ser executado, o nosso programa pode provocar um "crash" da máquina sem que saibamos exactamente o que correu mal.

9. *Determinação de erros*

Pode-se usar para este efeito um programa Debugger, cujo modo de usar já descrevi em geral. A determinação dos erros de um programa que se encontra na memória de um computador é não só extremamente importante para descobrir porque não funciona como ainda para nos ajudar a compreender o que se passa no interior da máquina.

10. *Gravação em cassette*

Como já indiquei, a gravação de um código-máquina não é o mesmo que gravar um programa Basic. É portanto importante que se saiba fazer ambas as coisas, a fim de dispor de uma cópia em memória permanente de todas as nossas rotinas.

11. *Execução*

Estamos finalmente na fase em que tudo parece estar correcto, e em que vamos tentar executar a rotina. Mesmo esta execução não é no entanto uma operação fácil ou imediata.

Comecemos a estudar todo este processo partindo do seu início, de modo a termos uma ideia de tudo o que se passa durante o desenvolvimento de um programa.

A ideia geral

Vamos escrever uma rotina em código-máquina que seja:

1. Acessível a partir da Basic
2. Capaz de comunicar com a Basic
3. Capaz de somar dois números de 16 bits
4. Capaz de subtrair um número de 16 bits de outro, permitindo o uso do resultado por um programa Basic.

Vejamos então agora um fluxograma que contenha todos estes pontos considerados.

Fluxograma geral

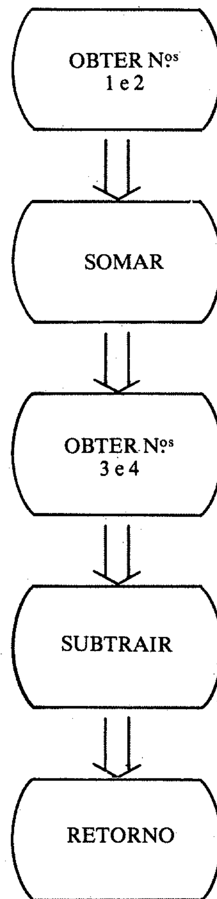


FIG. 18

Desenvolvimento da estrutura

O primeiro problema que encontramos consiste em como atribuir valores em Basic para uso de um programa em código-máquina, de tal modo que os números de 16 bits a somar ou subtrair sejam controlados pela Basic. Não podemos simplesmente atribuir um valor a uma variável Basic e esperar que o código-máquina a reconheça. É complicado transferir o conteúdo de uma variável Basic para um dos registos usados pela máquina; nestas condições, penso que esta solução não é a mais apropriada. Não esqueçamos ainda que não é possível carregar directamente um registo usando Basic, dado que não existem as instruções necessárias para tal nesta linguagem. Felizmente, no entanto, existe uma operação que pode ser realizada tanto em Basic como em dódigo-máquina: a transferência de um número para uma dada posição em memória, e a transferência seguinte deste valor para um registo em código-máquina ou para uma variável em Basic. Se queremos carregar um dado número numa posição de memória utilizando a Basic, utilizamos a ordem POKE. Para o transferir dessa posição para um registo usamos uma instrução LOAD de código-máquina. Estaremos então a colocar um valor num dado endereço, usando Basic, e a tirar em seguida esse valor para um registo usando código-máquina.

O problema seguinte consiste em transformar as respostas de código-máquina para uma forma que possa ser acedida em Basic.

A primeira solução possível é muito semelhante ao modo como colocámos anteriormente um valor Basic à disposição de um programa em código-máquina. Por outras palavras, podemos transferir a resposta obtida em código para um dado endereço, e depois ler esta resposta nesse endereço utilizando a ordem PEEK em Basic.

Por outro lado, se fizermos executar o código-máquina recorrendo à instrução PRINT USR xxxxx, onde "xxxxx" é a posição de memória onde se inicia o programa em código, o conteúdo do par de registos BC será impresso no visor quando a rotina termina. Isto significa que, se garantirmos que a resposta se encontra neste par de registos, bastar-nos-á a ordem PRINT para aceder ao resultado. Poderemos igualmente usar a instrução "LET A=USR xxxxx", onde "xxxxx" continua a designar o primeiro byte de código, dado que neste caso o conteúdo do par de registos BC será transferido para a variável A. Esta solução é de facto a mais prática, mas infelizmente só nos é possível guardar um único valor no par de registos BC. Em vez de voltarmos à Basic e reexecutar em seguida o código-máquina, convirá então usar uma combinação dos dois métodos. Por outras palavras, pode-

mos guardar a primeira resposta em endereços que possam ser acedidos pelo programa Basic, e carregar a segunda resposta em BC de modo a permitir a sua impressão ou a sua transferência para uma variável Basic.

Tendo decidido exactamente o modo como vamos transferir as constantes de Basic para código-máquina, e como vamos em seguida transferir as respostas de novo para Basic, devemos definir a sequência de operações a realizar entre estas duas operações. Vamos portanto decidir se convirá fazer primeiro a soma ou a subtracção. Esta questão não é muito importante, desde que nos recordemos de colocar o resultado da primeira rotina numa posição de memória específica e o resultado da segunda no par de registos BC. Decidi fazer a soma antes da subtracção; esta decisão não foi particularmente difícil de tomar — bastou lançar uma moeda ao ar...

Fluxograma — Versão 2

Apresento agora a segunda versão do fluxograma; observe-a cuidadosamente tendo em conta o que já foi dito:

Tirar os dois primeiros valores dos endereços onde se encontram.

Somar estes dois números.

Guardar o resultado numa posição de memória.

Tirar os dois valores seguintes dos respectivos endereços. Subtraí-los.

Guardar o resultado no par de registos BC.

Retorno ao comando Basic.

Programa Basic

Operações a executar em Basic:

Guardar os quatro valores.

Fazer executar a rotina em código-máquina.

Imprimir o resultado da segunda rotina.

Recuperar e apresentar o resultado das primeiras rotinas que foram carregadas em endereços de memória pelo programa em código-máquina.

Fim

Como somamos dois números de 16 bits?

Consideremos que os números acedidos na memória do computador foram carregados nos pares de registos HL e DE. O resultado da soma será então armazenado no par HL. É importante recordar que H é o registo mais significativo do par HL (H corresponde a "high", alto), e L é o registo menos significativo desse par (L corresponde a "low", baixo). O mesmo acontece em todos os outros pares de registos — D é o mais significativo e E o menos significativo do par DE. Tal como em qualquer soma normal, adicionamos primeiramente os dois bytes menos significativos — actuar de outro modo seria bastante problemático, o que se tornará óbvio para o leitor à medida que prosseguir o seu estudo.

Necessitamos portanto de somar o registo. E ao registo L. Infelizmente, como recordamos, só é possível somar registo ao registo A. Como resolvemos o problema? É simples. Basta-nos transferir o valor em L para o registo A, somando em seguida E a A. O resultado da soma ficará então neste registo, devendo depois ser transferido para L.

Já decidimos portanto quais serão as duas primeiras instruções do nosso programa em código-máquina. Primeiramente devemos "transferir L para A", o que é indicado pelo diagrama seguinte:

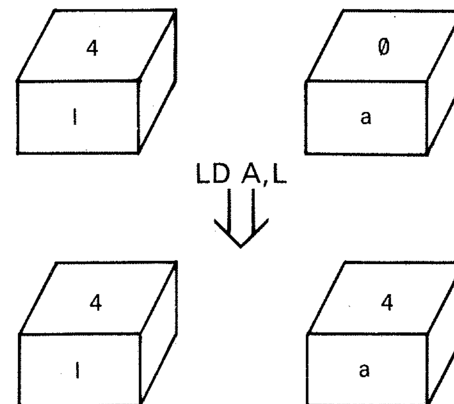


FIG. 19

Deveremos em seguida somar o registo E ao registo A; esta operação é novamente ilustrada por um diagrama:

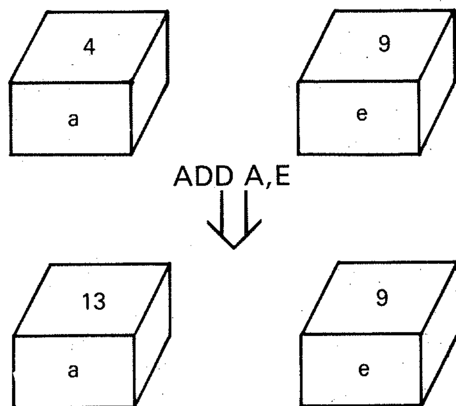


FIG. 20

Temos agora o resultado no registo A, mas não é aí que o queremos! Pretendemos que o resultado se encontre no par de registos HL, pelo que devemos transferir aquele valor para o registo L. Trata-se portanto de uma simples transferência do conteúdo de A para L:

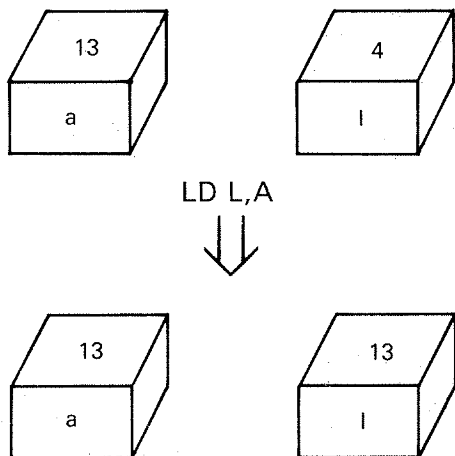


FIG. 21

Observemos agora rapidamente o que acontece no caso de a soma produzir um número superior a 255 — isto é, ao maior número que pode ser guardado em qualquer registo. Um aspecto interessante dos microprocessadores é que a resposta a este problema é perfeitamente previsível. O registo volta a contar a partir de zero! A soma 1+255 produz o resultado 0; a soma 20+255 produz o resultado 19. Mas em ambos os casos a flag "carry" é passado ao valor 1. Esta flag é um único bit do registo F, o qual é consultado pelo próprio processador a fim de saber se existe ou não transporte de uma unidade para um byte mais significativo. Se um número ultrapassou 255 há necessariamente um transporte, sendo a flag "carry" passada para o valor 1. Se não houver transporte, a flag voltará ao nível 0. Voltamos a este assunto porque se trata de uma questão essencial.

A própria flag "carry" é um tanto estranha: não pode ser acedida directamente, não permitindo portanto carregar em A, por exemplo, o estado em que ela se encontra. É no entanto possível controlar este estado, passando-o para 1 ou para 0 à nossa vontade. Ao somarmos os bytes mais significativos utilizaremos precisamente uma das ordens que têm em conta o estado da flag "carry". Utilizaremos uma instrução que faz o seguinte: soma E a H, e depois o resultado à flag "carry". Tendo sido a flag alterada pela soma dos bytes menos significativos, e não tendo sido afectada por qualquer transferência de registos, obter-se-á o resultado correcto.

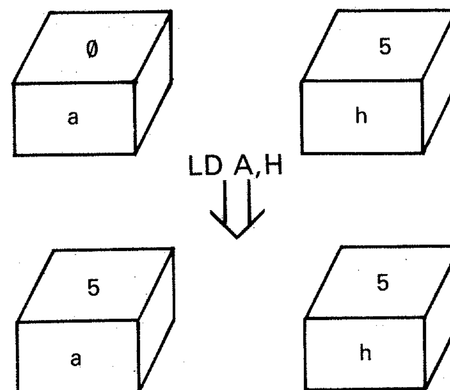


FIG. 22

Voltemos à nossa tarefa. Acabámos de somar os bytes menos significativos e estamos prontos a somar os mais significativos.

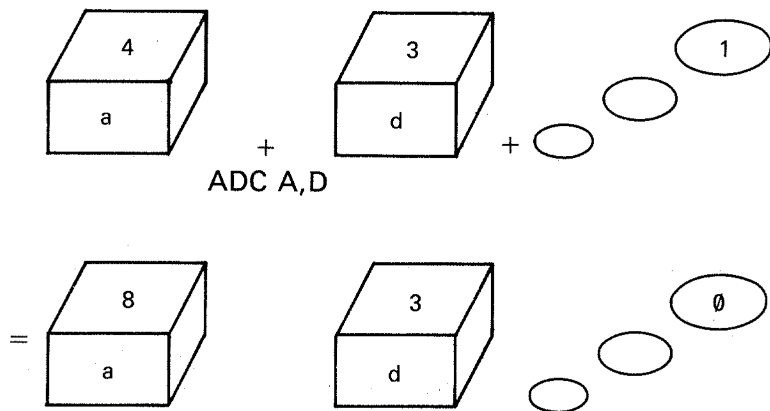


FIG. 23

Como não podemos somar D a H, mas apenas D a A, transferimos H para A, ou seja:

Em seguida somamos D ao registo A tendo em conta o estado da flag "carry", ou seja:

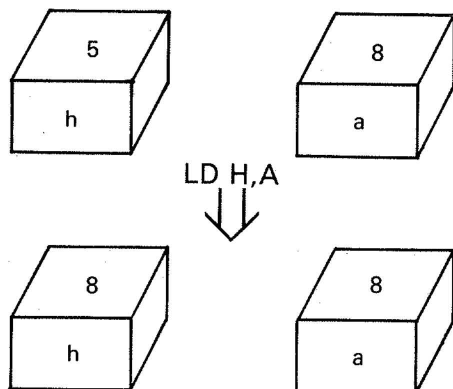


FIG. 24

A instrução que se segue é portanto "somar D e transporte a A".

Obtemos o resultado de D mais H mais a flag, o qual se encontra no registo A; interessa-nos agora transferi-lo de novo para H de modo a guardar o resultado final no par HL. A transferência é efectuada do seguinte modo:

A "soma" está terminada, mas deve agora ser acedida por ordens Basic. Podemos passar o resultado para BC de modo a imprimi-lo no visor, transferi-lo directamente para uma variável Basic, ou armazená-lo num endereço de memória a que a Basic possa aceder. Entre estas opções, a transferência para BC parece ser a melhor. No entanto, como desejaremos provavelmente utilizar BC para a segunda parte do programa, que executa uma subtracção, preferimos recorrer à outra solução. São necessárias duas posições de memória para guardar o resultado de uma soma executada em dois bytes (número de 16 bits). As posições onde a resposta pode ser carregada podem ser quaisquer, mas deve-se verificar se não se está a usar uma parte da RAM já usada por outro programa. Se tiver dúvidas, consulte o capítulo que trata do mapa de memória. Para os nossos objectivos, os endereços 27004 e 27005 serão óptimos. É possível distribuir a resposta por endereços que não sejam adjacentes, mas é inútil fazê-lo e dificulta a programação. Podemos agora instruir o programa para enviar o resultado para as posições de memória 27004 e 27005.

NOTA: O computador guarda o byte menos significativo antes do mais significativo, pelo que não nos devemos esquecer de multiplicar o conteúdo do segundo byte por 256.

Vamos agora tratar da subtracção de um registo de 16 bits por outro, usando uma ordem de subtracção de pares de registos. Como? Partamos do princípio de que os dois valores requeridos foram guardados nos pares de registos DE e HL, e que desejamos diminuir o valor de DE ao de HL — mantendo o resultado em HL. Basta-nos uma ordem para realizar isto, mas não esqueçamos que esta ordem tem em conta o estado da flag "carry". Se a última instrução executada levou esta flag para o valor 1, o resultado da subtracção pode ser afectado; convém portanto garantir que esta flag se encontra ao valor 0. É possível conseguir isto. Por agora utilizemos uma instrução falsa. "Passar a flag para zero", ou seja:

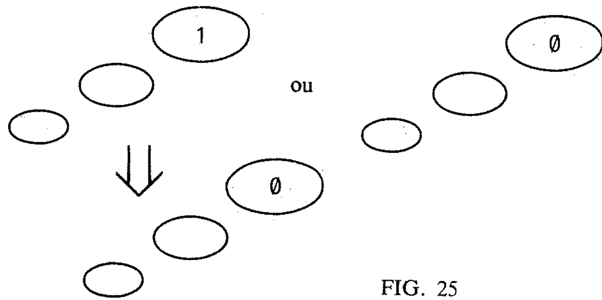
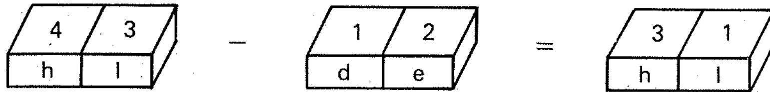


FIG. 25

Estando a flag "carry" no estado 0 já não poderá afectar o resultado da subtracção. Podemos portanto subtrair DE (com transporte) de HL — o que é indicado pelo diagrama seguinte:



O leitor terá assim subtraído o valor DE de HL, deixando o resultado neste último par de registos. Resta decidir como apresentar este resultado de modo a poder ser usado pela Basic. Evitámos anteriormente usar o par de registos BC para o caso de vir a ser necessário — talvez um modo trabalhoso de sublinhar este ponto, se bem que valha a pena sublinhá-lo. Agora é óbvio que BC pode ser usado. Transferimos H para B, e L para C. O resultado que originalmente se encontrava em HL passa portanto a estar em BC. Vejamos o diagrama correspondente.

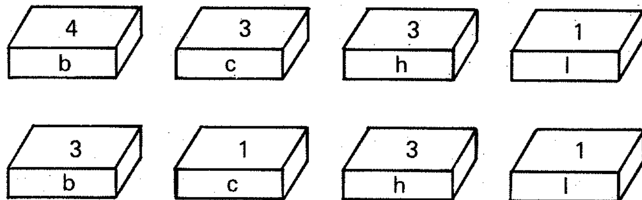


FIG. 26

O modo como o nosso programa deve funcionar está agora completamente sob controlo, e resta-nos transferir os valores originais Basic para os endereços de memória e levar o código a actuar sobre eles.

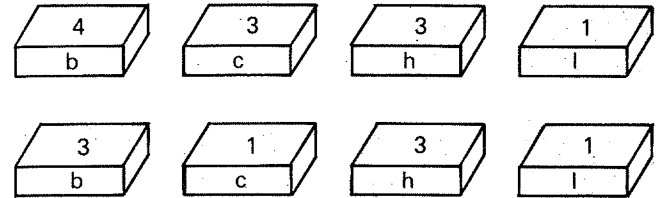


FIG. 27

Transferência de valores Basic para uso em código-máquina

O primeiro requisito consiste obviamente em converter os nossos números de 16 bits em pares de número de oito bits; estes podem ser colocados em memória e depois chamados pelo código. Trabalhando em hexadecimal, este processo é relativamente simples. Consideram-se os dois primeiros algarismos e determina-se o seu equivalente na "base 256", procedendo depois do mesmo modo em relação aos dois últimos algarismos. O resultado será formado pelas duas metades do número de 16 bits. No final deste livro o leitor encontrará uma tabela com todos os números entre 0 e 255 em decimal, hexadecimal e binário. Não hesite em consultar estas tabelas — aliviarão certamente o seu trabalho. Não se esqueça de que o computador guardará primeiro o byte menos significativo, e só depois o mais significativo. Proceda do mesmo modo. Se bem que isto não altere o resultado, desde que sejam feitas as alterações relevantes no programa, ajudá-lo-á a adquirir o hábito correcto.

Os oito números de 8 bits podem ser guardados em qualquer parte da RAM, desde que não interfiram com qualquer outra coisa. Já escolhi 27000 como ponto de partida. No caso da soma, portanto:

L = 27000
H = 27001

E = 27002
D = 27003

No caso da subtracção:

L = 27006
H = 27007
E = 27008
D = 27009

O resultado da soma será colocado nos endereços 27004 e 27005.

Tendo convertido os números de 16 bits em dois números de oito bits, é fácil colocar os números nos endereços escolhidos de modo a poderem ser usados pela rotina em código-máquina.

Como fazemos executar o programa em código-máquina

Não é possível usar GO TO relativamente a uma rotina em código-máquina, ou qualquer outra ordem Basic como por exemplo GO SUB. A Basic dispõe no entanto de uma outra ordem muito especial, a USR, para comandar a execução de programas em código-máquina. Em termos de Basic, USR não é uma ordem completa, sendo necessário utilizá-la com um dos seguintes prefixos:

PRINT, que imprime no visor o valor de retorno guardado no par de registos BC;

LET, que atribui a uma variável Basic o valor do conteúdo do par de registos BC;

RANDOMISE, que simplesmente define como ponto de partida da fórmula que determina números aleatórios o resultado contido no par BC.

Tanto PRINT como LET fazem gastar memória, e por esta razão usa-se geralmente RANDOMISE para aceder um código-máquina. Uma rotina USR assemelha-se bastante a uma rotina GO SUB, na medida em que constitui de facto uma subrotina; para sair dela é igualmente necessária uma ordem RETURN, que em termos de código-máquina é designada pela abreviatura RET.

Em seguida estudaremos o modo de aceder às constantes guardadas em bytes pelo programa Basic. O melhor modo de representar os números guardados nos respectivos endereços é recorrendo a um diagrama:

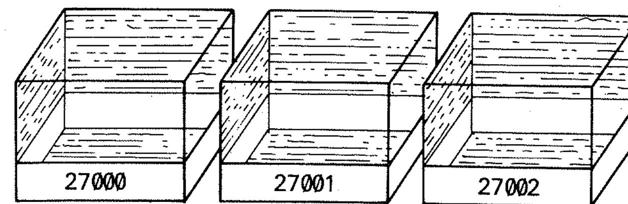


FIG. 28

Nestas condições, o passo seguinte consistiria talvez em usar uma ordem que carregasse o conteúdo do endereço 27000, por exemplo, no registo C. Note-se que isto envolve o uso de quatro rotinas lentas, pelo que deve ser desencorajado. Em vez disto recorremos à possibilidade de carregar um registo com o conteúdo de uma posição cujo endereço está guardado em HL. Usar este método, incrementando ou decrementando o conteúdo de HL a fim de seleccionar os bytes que devem ser carregados nos registos, é designado como "usar o registo HL como indicador" ("pointer"). O funcionamento de um "pointer" é esclarecido melhor no diagrama que se segue:

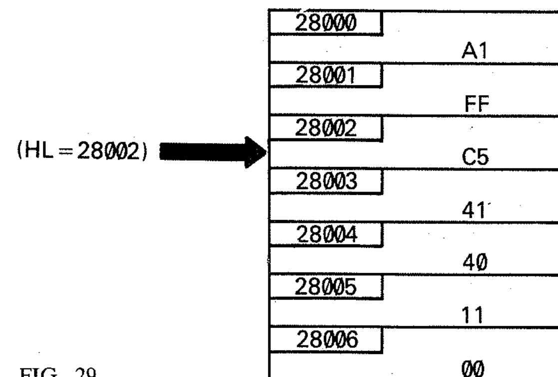


FIG. 29

Vejamos então o que se passa na prática.
Em primeiro lugar passa-se o "pointer" para 27000: "carregar em HL 27000".
Carrega-se a primeira metade do primeiro número de 16 bits em C: "transferir o conteúdo da posição HL para C".
Incrementa-se o "pointer": "Aumentar de 1 o valor de HL".
Carrega-se a segunda metade do primeiro número de 16 bits em B: "transferir o conteúdo da posição HL para B".
Carrega-se a primeira metade do segundo número de 16 bits em E: "transferir o conteúdo da posição HL para E".
Incrementa-se o "pointer": "Aumentar de 1 o valor de HL".
Carrega-se a segunda metade do segundo número de 16 bits em D: "Transferir o conteúdo da posição HL para D".
A transferência dos bytes está agora terminada, mas enfrentamos ainda o problema de carregar um dos valores em BC porque HL é usado como "pointer". HL é o único par de registos que pode ser usado para este efeito, dado que é o único registo que pode guardar um endereço a aceder. Utilizemos as seguintes duas instruções: "Transferir B para H", "transferir C para L". As constantes que foram definidas pelo programa Basic encontram-se agora nos pares de registos adequados.
Em seguida é necessário transferir as constantes Basic para realizar a subtracção. Note as grandes vantagens do sistema de "pointer". Um dos aspectos mais interessantes deste sistema consiste na facilidade de acesso pelo processador a diferentes endereços, alterando simplesmente o conteúdo inicial do registo HL — que está a usar como "indicador". Um outro aspecto do sistema de indicador é que possibilita não só o acesso a endereços seguidos para a "frente" como para "trás", se o conteúdo de HL for decrementado em vez de incrementado. Esta questão tornar-se-á mais clara através de uma descrição passo a passo do modo de transferir valores em memória para os registos.

Carregar no "pointer" o valor superior: "Carregar em HL 27009".
Transferir a segunda metade do segundo número de 16 bits para D: "Transferir conteúdo da posição HL para D".
Decrementar o conteúdo do "pointer": "Decrementar o conteúdo de HL".

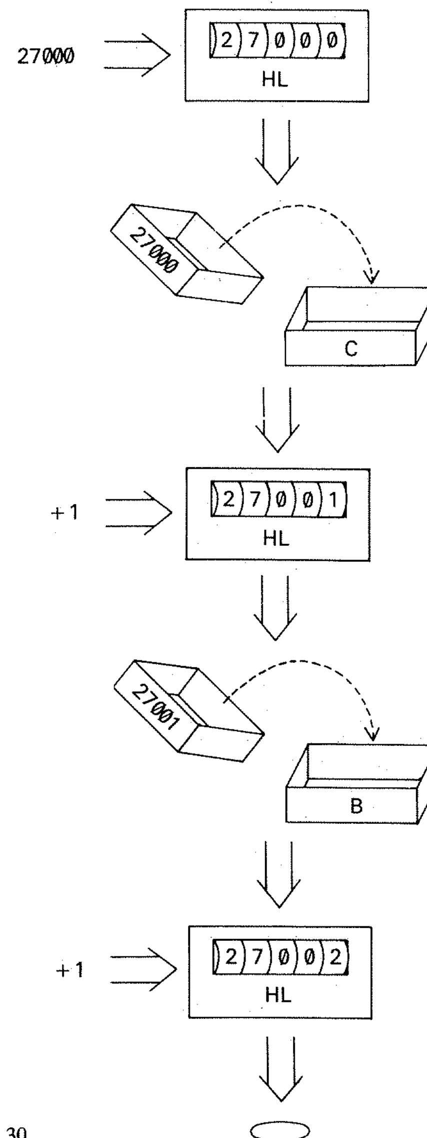


FIG. 30

Transferir a primeira metade do segundo número de 16 bits para E: "Transferir o conteúdo da posição HL para E".

Transferir a segunda metade do primeiro número de 16 bits para B: "Transferir o conteúdo da posição HL para B".

Decrementar o conteúdo do "pointer": "Decrementar o conteúdo de HL".

Transferir a primeira metade do primeiro número de 16 bits para C: "Transferir o conteúdo da posição HL para C".

Encontramos agora o mesmo problema de anteriormente: temos de transferir o conteúdo do par BC para HL — BC estava a ser usado para guardar temporariamente valores, enquanto HL servia de indicador de endereços. Portanto:

"Transferir B para H".
"Transferir C para L".

O diagrama da figura 31 mostra o que acontece.

O processo de transferência de dados entre o programa Basic e o programa em código está agora terminado. Os valores são processados e guardados novamente de um modo acessível em Basic. A escrita do programa está portanto quase terminada. Resta apenas transferir o comando da linguagem-máquina para a Basic.

A ordem USR, que utilizámos para transferir o comando para o código-máquina, já foi referida anteriormente, tendo-se dito que podia ser essemelhada a uma ordem GO SUB em Basic. Quando a subrotina encontra uma ordem RETURN em Basic, o processador volta à linha seguinte àquela onde a subrotina foi chamada. Por exemplo, se todas as linhas Basic estiverem numeradas e diferirem de 10, e a instrução GO SUB se encontrar na linha 10, a instrução RETURN reenviará a execução para a linha 20. Em código-máquina, usa-se também uma instrução "return" para transferir o comando da linguagem-máquina para a instrução Basic, que se segue àquela que chamou a rotina em código. Se se encontrar uma instrução USR na linha 10, e a instrução seguinte se encontrar na linha 20, quando o processador encontra a instrução RET na rotina em código-máquina devolve o comando à linha 20 do programa Basic.

Nestas condições, a última instrução de um programa em código-máquina deverá ser "retorno a Basic"

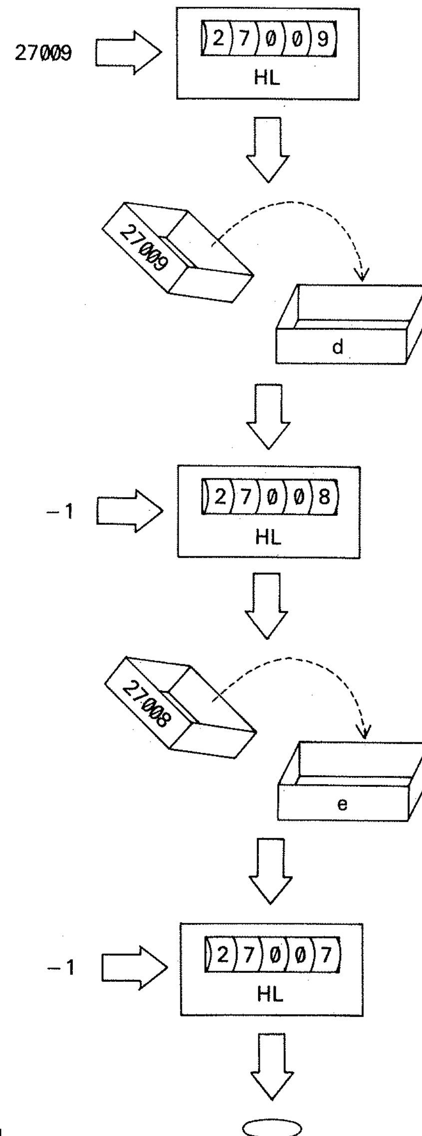


FIG. 31

Juntando tudo...

Convém, antes de prosseguirmos, fazer uma listagem clara e lógica das instruções que constituem o programa.

Primeiramente apresentaremos as instruções Basic necessárias para transferir os valores originais; depois o modo como funciona o programa em código-máquina, incluindo os movimentos dos dados e as rotinas de soma e subtracção. Finalmente indicaremos o modo de aceder em Basic as respostas obtidas.

BASIC:

Transferir o primeiro valor a somar para os endereços 27000 e 27001 (primeiro o byte menos significativo).

Transferir o segundo valor a somar para os endereços 27002 e 27003 (primeiro o byte menos significativo).

Transferir o primeiro valor a subtrair para os endereços 27005 e 27006 (idem).

Transferir o segundo valor a subtrair para os endereços 27007 e 27008 (idem).

Passar para a rotina em código-máquina.

CÓDIGO-MÁQUINA:

Carregar 27000 no par de registos HL.
Transferir o conteúdo da posição HL para C.
Incrementar o valor de HL.
Transferir o conteúdo da posição HL para B.
Incrementar o valor de HL.
Transferir o conteúdo da posição HL para E.
Incrementar o valor de HL.
Transferir o conteúdo da posição HL para D.
Transferir B para H.
Transferir C para L.
Transferir L para A.
Somar E e A.
Transferir A para L.
TransferirH para A.
Somar D com transporte a A.
Transferir A para H.
Transferir HL para as posições 27004 e 27005.
Ausência de operação.
Ausência de operação.
Carregar o par de registos HL com 27009.
Transferir o conteúdo da posição HL para D.

Decrementar o valor de HL.
Transferir o conteúdo da posição HL para E.
Decrementar o valor de HL.
Transferir o conteúdo da posição HL para B.
Decrementar o valor de HL.
Transferir o conteúdo da posição HL para C.
Transferir B para H.
Transferir C para L.
Passar a flag "carry" para 0.

Subtrair com transporte o par de registos DE do par de registos HL.

Transferir H para B.
Transferir L para C.
Retorno à Basic.

BASIC:

Imprimir valor final do par de registos BC (resultado da subtracção).

Obter o resultado da soma usando PEEK e imprimi-lo.

Conversão para linguagem Assembly

Aquilo que pretendemos do computador é agora bastante claro, e podemos portanto passar à conversão destas instruções para mnemónicas em linguagem Assembly, que serão depois introduzidas num programa Assembler. É igualmente boa ideia determinar neste momento o código hexadecimal das instruções, que podem também ser introduzidas através de um programa Monitor. Pode-se assim verificar o resultado. Em seguida apresentamos um quadro que inclui as instruções, as mnemónicas Assembly e o código hexadecimal:

Instruções	Linguagem Assembly	Hexadecimal
Carregar 27000 em HL	LD LH,27000	21 78 69
Conteúdo da posição HL para C	LD C,(HL)	4E
Incrementar valor de HL	INC HL	23
Conteúdo da posição HL para B	LD B,(HL)	46
Incrementar valor de HL	INC HL	23
Conteúdo da posição HL para E	LD E,(HL)	5E
Incrementar valor de HL	INC HL	23
Conteúdo de HL para D	LD D,(HL)	56

Transferir B para H	LD H,B	60
Transferir C para L	LD L,C	69
Transferir L para A	LD A,L	7D
Somar E a A	ADD A,E	83
Transferir A para L	LD L,A	6F
Transferir H para A	LD A,H	7F
Somar com transporte D a A	ADC A,D	8A
Transferir A para H	LD H,A	67
HL para 27004 e 27005	LD (27004),HL	22 7C 69
Ausência de operação	NOP	00
Ausência de operação	NOP	00
Carregar 27009 em HL	LD HL,27009	21 81 69
Conteúdo da posição HL para D	LD D,(HL)	56
Decrementar HL	DEC HL	2B
Conteúdo da posição HL para E	LD E, (HL)	5E
Decrementar valor de HL	DEC HL	2B
Conteúdo da posição HL para B	LD B, (HL)	46
Decrementar HL	DEC HL	2B
Conteúdo da posição HL para C	LD C, (HL)	4E
Transferir B para H	LD H,B	60
Transferir C para L	LD L,C	69
Limpar flag "carry"	AND 0	E6 00
Subtrair com transporte DE de HL	SBC HL,DE	ED 52
Transferir H para B	LD B,H	44
Transferir L para C	LD C,L	4D
Retorno à Basic	RET	C9

Execução "a seco"

É chegado o momento de verificar cuidadosamente o programa procurando quaisquer erros que possam existir. Se existir um erro grave é provável que o computador "estoire" sem lhe dar qualquer indicação sobre o que está errado. Depois de ter o trabalho de introduzir todo o programa na máquina não é nada agradável que tal aconteça. Existe sempre uma fase no desenvolvimento de um programa em que é conveniente ensaiá-lo "a seco".

Executar um programa "a seco" significa de facto executá-lo... no papel! Pode-se, por exemplo, construir uma tabela com todos os registos, escrevendo os valores que contêm depois de cada operação — tendo em conta o estado da flag "carry", etc. Os erros tornam-se normalmente óbvios quando se utiliza este método. Uma outra característica útil deste método de verificação é que cria um documento indicador daquilo que o programa faz, que permite mais tarde verificá-lo ou alterá-lo. Apresentamos na figura 32, um quadro muito simples que nos ajudará a

rever "a seco" os nossos programas. Consideremos cada mnemónica em linguagem Assembly, decidamos o que faz, e em seguida incluamos na tabela o resultado da instrução. Por exemplo, se encontrarmos a ordem LD A,40, escrevemos 40 na coluna A. Deste modo poderemos verificar imediatamente o conteúdo de cada registo. A extensão desta tabela de modo a incluir todas as posições de memória até agora usadas no nosso programa pode ajudar o leitor a familiarizar-se com o seu uso.

H	L	D	E	B	C	A	"Carry", se conhecida

FIG. 32

Introdução do programa no Assembler

Se bem que tenhamos listado os códigos hexadecimais, não é habitual fazê-lo quando se está a usar um Assembler, dado que é precisamente para isso que ele serve. Suponhamos que temos um Assembler, e o programa que desejamos converter.

Depois de carregar o Assembler deve-se dar-lhe as instruções de funcionamento requeridas. No caso do Assembler aqui referido será necessário dar-lhe a ordem "GO", indicando que se pretende converter um programa. Em seguida usa-se a instrução "ORG" juntamente com um endereço, identificando a posição de memória onde desejamos guardar o programa convertido. Se bem que estejamos já prontos a escrever as mnemónicas em declaração REM, vale a pena incluir um comentário para identificação do programa numa fase ulterior. Isto é feito incluindo um ponto de exclamação a seguir à declaração REM, e escrevendo em seguida o comentário. Neste caso seria apropriado escrever "rotina de soma e subtracção".

Cada mnemónica deve em seguida ser escrita cuidadosamente a fim de garantir que não haja erros. Verifique a lista final de mnemónicas pela lista original. É possível incluir mais do que uma instrução por linha, desde que sejam separadas por pontos e vírgulas, mas é muito mais fácil ler a rotina se apenas contiver uma instrução em cada linha. Vejamos a aparência que a lista de instruções deve ter depois de ser introduzida na máquina:

Versão final

```
ld hl,27000        217869
ld c,(hl)          4E
inc hl             23
ld b,(hl)          46
inc hl             23
ld e,(hl)          5E
inc hl             23
ld d,(hl)          56
ld h,b             60
ld l,c             69
ld a,l             7D
add a,e            83
ld l,a             6F
ld a,h             7C
adc a,d            8A
ld h,a             67
ld (27004),hl      227C69
nop                00
nop                00
ld hl,27009        218169
ld d,(hl)          56
dec hl             2B
ld e,(hl)          5E
dec hl             2B
ld b,(hl)          46
dec hl             2B
ld c,(hl)          4E
ld h,b             60
ld l,c             69
and 0              E600
sbc hl,de          ED52
ld b,h             44
ld c,l             4D
ret                C9
```

FIG. 33

Estamos quase prontos a converter a rotina para código-máquina, mas é ainda necessário cuidar de alguns pormenores.

Incluamos um comentário a seguir à última linha indicando que se trata do fim da rotina. As instruções "GO", "ORG" e "Finish" são muito importantes para o próprio Assembler, que aliás se recusará a funcionar se estiverem ausentes. A conversão será interrompida e será apresentada uma mensagem de erro. Se forem incluídos demasiados espaços ou um número insuficiente destes, o facto pode ser mal interpretado. Isto pode provocar um problema sério dado que constitui um erro difícil de detectar.

Depois de o programa ter sido convertido é aconselhável fazer uma cópia em cassette ou microdrive a fim de não perder o que já se fez no caso de o programa produzir um "crash". Para gravar o programa escreva:

SAVE "nome" CODE xxxxx,yyyyy

onde xxxxx indica a posição onde se inicia a rotina e yyyyy o número de bytes que a constituem. A instrução CODE é vital, pois indica ao computador que se trata de um programa em código-máquina. Para verificar a gravação escreva VERIFY """CODE. Para gravar o programa Basic que contém as mnemónicas, que é mais fácil de alterar a reconverter, use as instruções normais, ou seja:

SAVE "nome"

Para carregar o programa em código-máquina na sua forma hexadecimal, escreva LOAD "nome"CODE. A máquina carregará então um programa em código-máquina chamado "nome", colocando-o no endereço do qual foi originalmente gravado. Para recarregar um programa Basic, ou seja o programa que contém as mnemónicas em linguagem Assembley, escreva LOAD "nome".

O programa Basic usado para conter as mnemónicas antes de se realizar a conversão já não tem qualquer utilidade, podendo ser apagado. Não tente porém fazê-lo usando a ordem NEW — a menos que queira perder tudo o que está no computador — use CLEAR xxxxx, onde xxxxx é o endereço a partir do qual deseja "proteger" o que se encontra na máquina, menos um. Deste modo define uma nova RAMTOP, permitindo em seguida o uso da ordem NEW para apagar o que se encontra abaixo daquela.

Tanto o seu programa em código como o Assembler estarão entretanto seguros acima da RAMTOP.

Vejamos agora o programa de comando que organizará a transferência das variáveis para a memória e o uso dos resultados depois de o programa em código ser executado.

```
10 POKE 27000,4:POKE 27001,1:
   POKE 27002,3:POKE 27003,2
20 POKE 27006,7:POKE 27007,1:
   POKE 27008,8:POKE 27009,9
30 PRINT "SUB" = ";USR 28000
40 PRINT "SOMA = ";PEEK 27004 +
   (PEEK 27005)*256
50 STOP
```

É muito prático poder carregar simultaneamente na máquina o programa de comando Basic e o código-máquina. Para tal faz-se o programa Basic entrar em auto-execução a partir de uma primeira linha que carrega o código. Pode-se então gravar ambos os programas seguidos na cassette.

Depois de verificar uma cópia do programa, resta apenas executá-lo, o que se consegue recorrendo à instrução RUN ou GO TO. Poderá então experimentar diferentes valores nos registos, tentando prever os resultados.

SALTAR É DIVERTIDO

VIII

INSTRUÇÕES DE SALTO

Durante todo este livro preocupámo-nos essencialmente com o desenvolvimento de uma técnica "correcta". É chegado o momento de estender o reportório das ordens disponíveis para uso em programação — o que é evidentemente necessário para que os seus instintos criativos se exprimam completamente. Quanto mais sabemos, maior é a quantidade de recursos à nossa disposição. Tornar-se-á cada vez mais aparente que é possível obter muitas combinações a partir de apenas algumas ordens, aumentando portanto a versatilidade dos programas. Este capítulo tratará de saltos simples, saltos condicionados, saltos relativos, saltos condicionados relativos, chamadas de subrotinas e chamadas condicionadas. Além disto falaremos do registo PC, da instrução RET, e de outros aspectos do funcionamento interno do computador associados a estas instruções.

O registo PC

A organização dos registos da CPU já foi referida em termos genéricos, tendo sido dada alguma atenção aos mais fáceis de manipular (HL, BC, etc.). Existem porém outros pares de registos que não podem ser acedidos directamente ou que só o podem ser para operações relativamente básicas. Isto resulta da necessidade da CPU reservar espaço para armazenamento das suas próprias variáveis, onde estas possam ser acedidas e se encontrem livres de interferências externas. O registo PC é um bom exemplo do que acabamos de dizer. Trata-se de facto de um par de registos que contém o endereço da posição em memória da ordem em

execução e que portanto actua como "pointer" (indicador) da CPU, indicando-lhe a posição da instrução seguinte.

Quando se liga o computador o valor do registo PC é 0000, pelo que a primeira ordem executada pela CPU é a situada na primeira posição de memória. É por isto que a ROM Sinclair ocupa os primeiros 16 K de memória. Por outras palavras, através do uso de uma pastilha integrada ROM, os valores das posições 0 a 16383 são previamente definidos, e nestas condições quando a máquina é ligada executa imediatamente uma rotina de inicialização. Quando está a ser executada uma rotina em código-máquina o registo PC contém sempre o endereço da instrução actual, de tal modo que quando esta é terminada o conteúdo do PC altera-se em função do comprimento da instrução. Passa portanto a conter o endereço da instrução seguinte. Pode-se assim considerar que, se o conteúdo deste registo for alterado externamente, se torna possível alterar o endereço onde a CPU vai procurar a instrução seguinte.

O primeiro problema a vencer é que não existem ordens que alterem directamente o conteúdo do registo PC, ao contrário do que acontece com os registos HL ou BC, por exemplo. Felizmente existe uma instrução que altera o conteúdo do registo PC, a saber a instrução JP (de "jump", salto). Esta instrução assemelha-se bastante à GO TO da linguagem Basic, e funciona carregando simplesmente o registo PC com o valor pretendido — o do endereço onde deve ser lida a instrução seguinte. Em Basic, se estivermos na linha 100 e quisermos fazer executar uma instrução na linha 1050, usaremos uma instrução GO TO 1050. Em código-máquina a operação é exactamente a mesma. Segue-se daqui que, se tivermos uma rotina curta nas posições 30000 a 30012, e quando esta rotina tiver terminado quisermos executar uma outra guardada a partir do endereço 32000, teremos de dar à máquina uma instrução de salto do seguinte tipo:

```
30000 — 300012      : primeira rotina
30012 JP 32000      : salto para endereço 32000
32000 —             : segunda rotina
```

Mostra-se a seguir um exemplo desta instrução:

```
org — 30000
```



```
30000 3E 00          Ld a,0
30002 06 07          Ld b,7
30004 C3 00 7D       JP 32000
32007 60             Ld h,b
32008 6F             Ld l,a
32009 C9             ret
```

Se bem que seja obviamente útil poder saltar de uma parte da memória para outra, e executar as instruções dessa parte da memória, em quantos casos será necessário escrever uma rotina em duas zonas de memória separadas? Não seria muito mais fácil executar uma rotina num endereço longínquo apenas se uma dada condição for ou não verdadeira? Por exemplo, num programa Basic incluindo cálculos complicados, pede-se ao computador que indique se um resultado é maior ou menor do que um termo de comparação usando uma rotina do seguinte tipo:

```
 500 IF resposta maior do que previsto THEN GO TO 1000
 510 IF resposta inferior ou igual ao previsto THEN GO TO 1500
1000 PRINT "Maior do que"
1010 STOP
1500 PRINT "Menor ou igual a"
1510 STOP
```

Note que neste programa Basic se usou a declaração STOP para terminar a rotina e que não existe qualquer ordem semelhante em código-máquina — em particular porque nunca é necessário que o computador páre completamente. Quando é constituído um programa em código-máquina, é sempre necessário um retorno à Basic. Em uso normal a instrução RET reenvia o comando para esta linguagem.

Em seguida apresento uma lista das seis instruções de salto condicional mais usadas:

JP z	: Saltar se zero
JP nz	: Saltar se não zero
JP c	: Saltar se há transporte
JP nc	: Saltar se não há transporte
JP m	: Saltar se menos
JP p	: Saltar se positivo

JP z significa que o salto só será realizado se a flag "zero" estiver ao nível 1, ou seja, se o resultado do cálculo anterior foi zero. Se o resultado tiver sido diferente de zero, e portanto a flag for igual a zero, a máquina não executa o salto e continua para a instrução seguinte.

Por exemplo:

```
org 28000
28000 3E 00         ld a 0
28002 FD 69         add 0
28004 CA 6B 6D      jp z,28011
28007 3D            dec a
28008 C3 62 6D      jp 28002
28011 C9            ret
```

Neste programa pretende-se determinar se a soma de zero ao acumulador produz um resultado nulo. Se assim for, o comando é reenviado à Basic; se não, será decrementado o valor do acumulador. Este conterá portanto o mesmo número menos um, sendo novamente realizado o teste sobre este valor. Este processo continuará até o acumulador conter o valor zero. Deve-se verificar sempre se a instrução usada antes de um salto condicional altera de facto o estado da flag usada para o teste; quando tiver dúvidas, consulte a última tabela deste livro.

JP nz significa que o salto só será efectuado se a flag "zero" estiver ao nível zero, isto é, se o resultado da última operação efectuada não for zero. Se, como no exemplo anterior, a instrução anterior for "somar zero", seguida por JP nz,32000, o código-máquina terá a seguinte aparência:

```
29999 3D            DEC A
30000 FD 69         ADD 0
30002 CA "fim"      JP NZ,29999
30005 C3 "32000"    JP 32000
```

O equivalente Basic destas instruções seria:

IF A + 0 = 0 THEN GO TO 32000

JP c significa "saltar apenas se a flag 'carry' (transporte) estiver ao nível 1". Por outras palavras, se o último cálculo efectuado produziu em excesso e a necessidade de transportar uma unidade para o algarismo mais significativo seguinte, a flag "carry" terá passado para o valor 1 e a condição será cumprida. Um exemplo simples consistiria em levar a execução a saltar para um endereço novo no caso de os registos A e B, em conjunto, darem um valor superior a 255. Apresento em seguida um programa que executa este teste:

```
org 30000
30000 80            add a,b
30001 DA 30 75      jp c,30000
```

O equivalente Basic seria: IF A mais B é maior do que 255 THEN GO TO 30000. É assim possível verificar erros que ocorram quando a soma de duas variáveis é um valor excessivamente grande para poder ser manipulado por um registo.

A instrução JP nc é muito semelhante à anterior, exceptuando o facto de o salto ser apenas executado quando a flag "carry" está ao nível zero. Verifica se o resultado da última operação produziu um transporte; se não — ou seja, se a soma dá um resultado inferior a 256 — a execução salta para o endereço indicado. Vejamos um exemplo:

```
org 30000
30000 80            add a,b
30001 D2 30 75      jp nc,30000
```

O equivalente Basic seria:

IFA + B<256 THEN GO TO 30000

As duas últimas condições de que dispomos para determinar os saltos são a "menos" e a "positivo". Têm a ver com o facto de a flag de sinal reflectir um valor positivo ou negativo. Como ainda não estudámos esta flag, bastará dizer que JP P significa "saltar" se a flag de sinal reflectir um número positivo. A ordem JP M significa "saltar se a flag de sinal reflectir um número negativo".

Ao longo dos exemplos apresentados usaram-se endereços para demonstrar precisamente o que está a acontecer, mas se se usar um Assembler ou se escrever um programa que possa vir a ser colocado em qualquer posição de memória torna-se bastante mais fácil usar "etiquetas". Para executar uma rotina e voltar ao princípio, etiqueta-se o início do programa com a palavra INICIO, por exemplo, e no final introduz-se a ordem JP INICIO. Usando um Assembler não há necessidade de usar endereços directos, pois esse será o trabalho do próprio Assembler, que determinará os endereços de cada parte do programa. É igualmente possível etiquetar certos bytes especiais no interior do programa que podem ser usados para guardar respostas ou resultados de cálculos durante a execução da rotina. Esta função é uma das mais úteis do Assembler, além da própria conversão das mnemónicas. Apresenta-se em seguida um curto exemplo do modo de usar etiquetas. Não se preocupe com a forma como o programa funciona, que lhe parecerá talvez um pouco obscura; tome nota das etiquetas e da forma como são usadas e definidas.

1. Listagem Basic para o Assembler ACS.

```
  1 REM go
  2 REM org 30000
 10 REM ld hl,22527
 20 REM ld b,192
 30 REM !Alterar B e HL para diferenciar blocos do visor
 40 REM A; ld c,32
 50 REM B; ld e, (hl)
 60 REM rl (hl)
 70 REM dec hl; dec c
 80 REM jr nz,B
 90 REM bit 7,e
100 REM jr z,C
110 REM push hl; ld de,32
120 REM add hl,de
130 REM set 0, (hl)
140 REM !Res 0,(HL) para não reintroduzir os pixels
150 REM pop hl
160 REM C; and a
170 REM djnz,A
180 REM ret
190 REM finish
```

2. Listagem do programa convertido

```
org 30000
30000 21 FF 57          ld hl,22527
30003 06 C0             ld b,192
Alterar B e HL para diferentes blocos de visor
A
30005 0E 20             ld c,32
B
30007 5E                ld e, (hl)
30008 CB 16             rl (hl)
30010 2B                dec hl
30011 0D                dec c
30012 20 F9             jr nz,B
30014 CB 7B             bit 7,e
30016 28 08             jr z,C
30018 E5                push hl
30019 11 20 00          ld de,32
30022 19                add hl,de
30023 CB C6             set 0, (hl)
Res 0,(HL) para não reintroduzir os pixels
30025 E1                pop hl
C
30026 A7                and a
30027 10 E8             djnz,A
30029 C9                ret
```

É evidentemente possível saltar para uma posição anterior do programa. Se estivermos na posição 30000 e quisermos executar uma rotina em 28000, é perfeitamente legítimo usar a instrução JP 28000. É necessário ter em conta um factor muito importante quando se incluem saltos que reenviam para ciclos que terminam neles próprios, e que é o uso de saltos incondicionais que não possuem instruções RET entre si. Trata-se do modo mais rápido de colocar o computador num ciclo de execução eterno — sem qualquer possibilidade de fuga! É possível fugir a estes ciclos em Basic carregando em BREAK, mas a execução em código-má-

quina não é sensível a esta tecla. Alguns computadores permitem parar a execução de um código sem perder todas as rotinas presentes em memória, mas isto não é possível em qualquer dos computadores Sinclair. O único modo de sair de um ciclo eterno consiste em desligar o computador... Enquanto executa um código-máquina o Spectrum não lê a tecla Break, ou aliás qualquer outra, a menos que no próprio programa esteja inserida uma rotina para leitura do teclado.

Estudámos já os saltos directos que nos permitem carregar o registo PC, ou Contador de Programa, com um novo endereço ao qual o processador irá buscar a instrução seguinte. Existe um outro tipo de instrução de salto — útil quando se escrevem curtas rotinas que não dependem da área da memória onde se encontram.

Este novo tipo de salto é conhecido pelo nome de salto relativo. A razão deste nome é que permite saltar para uma posição distante da actual de um certo número de bytes para a frente ou para trás. Em vez de saltar, por exemplo, do endereço 28000 para o 28005, pode-se usar um salto relativo de mais cinco, ou de menos cinco no caso inverso. Isto pode parecer um modo excessivamente complicado de saltar de uma posição para outra, mas apresenta de facto a vantagem de permitir guardar a rotina em qualquer posição da memória. Ao usar saltos relativos, dado que não se usam endereços definidos para a instrução JUMP, pode-se passar a rotina da posição 28000 para a 24000 ou qualquer outra sem necessidade de introduzir qualquer alteração no código.

Esta possibilidade é particularmente útil quando a rotina pode ser usada mais do que uma vez no interior de um programa maior, ou quando se converte um programa para uso num computador de memória mais extensa. Do mesmo modo, pode ser muito útil quando não existe espaço suficiente em memória para uma dada rotina, sendo necessário portanto deslocá-lo para um endereço inferior para caber toda. A ordem de salto relativo pode poupar muito trabalho mais tarde. Existe uma limitação grande nesta ordem, dado que só é possível saltar para uma posição que se encontre a um máximo de mais 129 bytes ou menos 126 bytes relativamente ao endereço actual. Isto pode parecer muito limitativo, mas na prática esta margem é mais do que suficiente em rotinas normais. O cálculo de deslocamentos superiores a este é complicado. Basta dizer que a instrução de salto relativo (JR) deve ser considerada como uma instrução de salto local.

O cálculo do deslocamento de um salto relativo é razoavelmente fácil, se bem que deva ser feito com cuidado. Não é possível usar directamente números negativos no interior de um programa em código-máquina. No caso de um deslocamento positivo para um salto relativo, ou seja entre 0 e 129, deve-se indicar ao computador o valor desejado. Quando o deslocamento é negativo, ou seja entre 0 e menos 126, deve-se indicar ao computador o valor 256 menos o deslocamento desejado (para um deslocamento de -10, indica-se ao computador 246). Usando o Assembler indicado, basta escrever o deslocamento positivo ou negativo, não sendo necessário preocuparmo-nos com o cálculo do valor a indicar à máquina. Note-se por outro lado que os deslocamentos variam de facto na gama -128 a +127, mas são contados a partir do byte que se segue ao fim da instrução de salto; noutros Assemblers é necessário ter em conta este facto.

Ao executar um salto relativo, o computador é instruído para saltar para um byte diferente do que se segue. Por outras palavras, se o computador executar a instrução JR 0 (salto relativo de zero bytes) limitar-se-á a passar para a instrução seguinte, dado que terá muito simplesmente aumentado o valor do registo de PC de zero. No entanto, como o comprimento da instrução de salto relativo é de dois bytes (um para a instrução propriamente dita e outro para o deslocamento), a instrução JR-2 reenviará a execução para a própria instrução de salto, resultando num ciclo infinito.

Apresenta-se em seguida um exemplo do uso da ordem de salto relativo, que ilustra a utilidade do uso de etiquetas.

```
org 30000
Start
30000 3E 29          ld a,41
30002 06 28          ld b,40
30004 18 06          jr Part2
30006 00             nop
30007 00             nop
30008 00             nop
30009 00             nop
30010 00             nop
30011 00             nop
```

```
Part2
30012 0E 0E          ld c,14
30014 FD 69          add c
30016 18 02          jr +2
30018 18 EC          jr Start
30020 C9             ret
```

É interessante notar que em momento algum se usam neste programa endereços definidos; a rotina pode ser de facto colocada em qualquer área da memória sem alterações.

É possível utilizar saltos relativos condicionados do mesmo modo que os saltos "absolutos". A única diferença reside em existirem menos tipos de condições para uso em saltos relativos. São apenas possíveis as seguintes:

```
JP z    : salto relativo se a flag zero igual a 1
JP nz   : salto relativo se a flag zero igual a 0
JP c    : salto relativo se a flag "carry" igual a 1
JP nc   : salto relativo se a flag "carry" igual a 0
```

Estes saltos podem ser usados da mesma maneira que as correspondentes versões de saltos "absolutos", bastando usar um deslocamento em vez do endereço directo.

Em Basic, além de termos a instrução GO TO, dispomos ainda da instrução GO SUB. Esta permite-nos executar uma rotina e voltar ao programa principal sem necessidade de recorrer a uma nova instrução GO TO. É assim facilitado o uso de subrotinas curtas. Uma vantagem disto é que é possível reutilizar uma dada rotina muitas vezes no interior de um programa. Por exemplo, realizar uma operação específica três ou quatro vezes no interior de um programa obrigaria a repetir o conjunto de instruções que a constituem o número de vezes necessário; mas é possível escrevê-la um só vez como subrotina.

Esta subrotina pode encontrar-se antes ou depois do programa principal, e pode ser acedida por uma ordem GO SUB em Basic. Quando terminada, a instrução RETURN no final da rotina leva o processador a reenviar o comando de execução para a instrução que se segue à GOSUB no programa principal. Pode-se actuar de modo semelhante em código-máquina; para chamar uma subrotina em código a partir de um programa também em código usa-se a instrução CALL, seguida pelo endereço (ou etiqueta se se está a usar um Assembler) da subrotina.

Mostra-se a seguir como é possível usar este procedimento:

```
  10 RANDOMIZE USR 30000
  org 30000
Start
  30000 CD 6E 6D          call,28014
  30003 C9                ret

  org 28014
Subrt
  28014 3E 00             ld a,0
  28016 06 00             ld b,0
  28018 C9                ret

  20 PRINT "ROTINA TERMINADA"
```

Pode-se ainda usar a instrução CALL para executar subrotinas já situadas na memória ROM. Se bem que muitas das rotinas no interior da ROM tenham a ver com a linguagem Basic e outras com o sistema operativo, existem ainda bastantes que se podem tornar úteis. Algumas destas rotinas são curtas, e a única razão para as chamar consiste em poupar tempo em vez de copiá-las para a RAM e incluí-las nos nossos programas.

Outras porém são compridas e complicadas, ganhando-se então bastante tempo e espaço de memória ao utilizá-las. O uso da instrução CALL diminui por outro lado as possibilidades de erro.

Depois de uma subrotina ter sido executada é necessário voltar ao programa principal. Em Basic isto é feito usando a instrução RETURN, em código-máquina usando a instrução RET. Isto pode parecer um tanto confuso, dado que a instrução RET já foi usada para reenviar o comando para a Basic no fim da execução de uma rotina em código-máquina; de facto, porém, a instrução RET é muito útil porque reenvia sempre o comando para o programa que chamou a rotina. Se o código foi executado através de uma instrução USR Basic, a RET reenvia o comando para a Basic. Se no entanto foi executado a partir de uma instrução CALL, reenvia o comando para o programa principal em código-máquina.

Depois de ter sido executada uma CALL, a posição onde esta se encontra é guardada pelo computador de tal modo que ao encontrar uma instrução RET o processador reenvia para o endereço apropriado. Depois de voltar ao programa principal, o valor da posição guardado pelo computador é substituído pelo valor anterior. Por outras palavras, quando se passa ao programa em código-máquina através da instrução USR, o endereço guardado pelo computador é o da instrução Basic seguinte.

Quando se passa a uma subrotina em código através de uma instrução CALL, o endereço desta instrução é igualmente guardado pelo computador. Quando é encontrada a instrução RET, volta-se ao endereço em causa, ou seja, ao programa principal. Como a subrotina já estará terminada, o valor guardado pelo computador passará a ser o que já existia antes de a subrotina ter sido chamada; ou seja, o endereço da instrução seguinte. O diagrama que se segue esclarece um pouco melhor o que se passa.

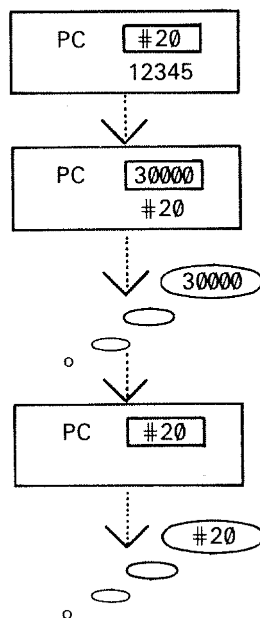


FIG. 34

É também possível usar instruções de chamada (Call) condicionais, tal como acontece nos saltos absolutos e relativos. Essas instruções funcionam do modo já explicado, não sendo necessário determo-nos mais tempo neste assunto. No caso das instruções CALL o uso de etiquetas aumenta igualmente a facilidade de construção do programa, ajudando ainda a estruturá-lo. Torna-se possível dividir um programa numa série de subrotinas acedidas por um programa principal bastante curto. Trabalhando deste modo consegue-se obter uma listagem mais compreensível, facilitando-se ainda a correcção e expansão do programa. Infelizmente não existem CALLs relativas, sendo portanto necessário usar sempre endereçamento directo — pelo que o programa não pode ser guardado indiferentemente em qualquer área da memória. Mesmo assim, porem, não é particularmente difícil colocá-lo em endereços diferentes porque, como o leitor recorda, não é necessário especificar o endereço ao qual é feito o retorno depois de cada subrotina ter sido terminada. Esta pode portanto ser colocada onde se quiser na memória, introduzindo-se depois o programa principal onde for conveniente e com as alterações devidas (indicando os endereços das subrotinas). As mnemónicas e os códigos decimais e hexadecimais destas instruções são indicados no Apêndice B.

CONTINUANDO A VIAJAR NO ZX SPECTRUM

IX

COMO USAR O TECLADO E O VISOR

Este capítulo tratará principalmente do modo como o computador e o utilizador podem "comunicar" entre si. Preocupar-nos-emos portanto essencialmente com o teclado e o visor. Começaremos por estudar a disposição das teclas e o modo como podemos usar as variáveis de sistema para ler facilmente o teclado. Em seguida veremos alguns exemplos e rotinas que facilmente poderão ser introduzidas nos seus programas. Veremos ainda como é composto o visor e qual a sua posição na memória. Não nos é difícil alterar o conteúdo da imagem actuando directamente sobre a área RAM onde esta se encontra armazenada, se bem que Clive Sinclair não nos tenha facilitado muito a vida "arrumando" a imagem de uma forma um tanto estranha.

Estudaremos o modo de ultrapassar algumas dificuldades inerentes à concepção do Spectrum, e a maneira de executar certas tarefas como a definição de caracteres novos e a apresentação subsequente destes no visor. Finalmente mostrar-se-á uma rotina em código-máquina que permite imprimir qualquer ponto no visor, que devido à sua enorme velocidade será de facto muito mais útil do que possa parecer à primeira vista. O teclado encontra-se dividido em linhas e colunas, o que permite ao computador identificar facilmente a tecla que é premida. Isto permite igualmente ao computador registar o facto de duas teclas diferentes serem premidas simultaneamente. Exemplos disto são as teclas CAPS SHIFT e SYMBOL SHIFT. Se porém estas teclas se encontrassem juntas, a situação poderia confundir o computador, que se tornaria incapaz de diferenciar entre estas e certas outras teclas premidas simultaneamente.

O modo como o computador lê o teclado é um tanto complicado, se bem que seja útil e interessante compreender o arranjo

das teclas. Felizmente para nós, não necessitamos de compreender ou sequer de usar a rotina em ROM que lê o teclado, devido à existência de duas variáveis muito úteis que o computador mantém continuamente na área de variáveis de sistema. Esta área começa no endereço 23552 e termina em 23665. Contém diversas variáveis, incluindo as que especificam as cores usadas, a duração dos sons, etc., e em particular as teclas que são premidas. Usando a função PEEK sobre o endereço 235557 é de facto possível descobrir se se carregou ou não numa tecla. Para demonstrar isto, escreva o pequeno programa Basic que se segue:

```
  10 IF PEEK 23557=5 THEN PRINT
AT 0,0; "Tecla premida"
  20 IF PEEK 23557<>5 THEN PRINT
AT 0,0;"              "
  30 GO TO 10
```

Este programa determina se se encontra ou não o valor 5 no endereço 23557, e se tal acontecer imprime a mensagem "Tecla premida". Tire o dedo da tecla; aquele endereço deixará de conter o valor 5 e a mensagem será apagada. A instrução GO TO 10 na linha 30 serve para manter o programa em execução indefinidamente.

CODE		CHR$	CODE		CHR$
32	=		33	=	!
34	=	"	35	=	#
36	=	$	37	=	%
38	=	&	39	=	'
40	=	(	41	=	)
42	=	*	43	=	+
44	=	,	45	=	—
46	=	.	47	=	/
48	=	0	49	=	1
50	=	2	51	=	3
52	=	4	53	=	5
54	=	6	55	=	7
56	=	8	57	=	9
58	=	:	59	=	;

60	=	<	61	=	=
62	=	>	63	=	?
64	=	@	65	=	A
66	=	B	67	=	C
68	=	D	69	=	E
70	=	F	71	=	G
72	=	H	73	=	I
74	=	J	75	=	K
76	=	L	77	=	M
78	=	N	79	=	O
80	=	P	81	=	Q
82	=	R	83	=	S
84	=	T	85	=	U
86	=	Y	87	=	W
88	=	X	89	=	Y
90	=	Z	91	=	[
92	=	\	93	=	]
94	=	↑	95	=	—
96	=	£	97	=	a
98	=	b	99	=	c
100	=	d	101	=	e
102	=	f	103	=	g
104	=	h	105	=	i
106	=	j	107	=	k
108	=	l	109	=	m
110	=	n	111	=	o
112	=	p	113	=	q
114	=	r	115	=	s
116	=	t	117	=	u
118	=	v	119	=	w
120	=	x	121	=	y
122	=	z	123	=	{
124	=	|	125	=	}
126	=	~	127	=	©
128	=		129	=	
130	=		131	=	
132	=		133	=	
134	=		135	=	
136	=		137	=	
138	=		139	=	
140	=		141	=	

142	=		143	=	
144	=	A	145	=	B
146	=	C	146	=	D
148	=	E	149	=	F
150	=	G	151	=	H
152	=	I	153	=	J
154	=	K	155	=	L
156	=	M	157	=	N
158	=	O	159	=	P
160	=	Q	161	=	R
162	=	S	163	=	T
164	=	U			

Esta informação é muito útil, mas não identifica a tecla em que se carregou, o que no entanto é feito pela variável de sistema localizada no endereço 23560. Esta posição é muitas vezes designada "Last K" (última tecla) porque contém o valor da última tecla premida e guarda este número mesmo depois de se deixar de carregar nela. Usando esta variável juntamente com a anterior pode-se descobrir se o utilizador carregou numa tecla, e em qual — começando obviamente pela primeira. Em seguida apresenta-se um curto exemplo Basic do que pode acontecer se apenas se usar a variável de sistema 23560, ou seja, o valor da última tecla premida:

```
10 PRINT PEEK 23560,: GO TO 10
```

Experimente isto e verificará que não é de grande utilidade; mas corrija o programa da seguinte forma:

```
 10 IF PEEK 23557=5 THEN PRINT
AT 0,0; CHR$ (PEEK 23560)
 20 GO TO 10
```

Note que para imprimir o caracter correspondente à tecla premida é necessária usar a instrução CHR$ aplicando-a ao conteúdo da variável de sistema. O computador guarda apenas o código correspondente ao caracter; como utiliza para isto apenas um byte, pode de facto armazenar qualquer um de 256 códigos diferentes.

Esta informação pode ser usada para escrever uma rotina em código-máquina que funciona de uma forma bastante semelhante à Basic, mas que carrega o valor da última tecla premida no registo C — apenas dando o valor no entanto se efectivamente tiver sido premida uma tecla. Vamos portanto dar uma utilização bastante eficaz à instrução de salto relativo condicionado. Devemos porém ter cuidado — como estamos em código-máquina, e como estamos a ler cada tecla separadamente, o computador não registará a tecla Break. Se não se carrega em qualquer tecla a máquina continuará eternamente em código-máquina. Esta situação pode não ser grave se soubermos que vai ser premida uma tecla e que voltaremos ao comando Basic depois de tal acontecer — depois do que é possível utilizar a tecla Break. No entanto, se utilizarmos esta rotina no interior de um programa completamente escrito em código-máquina será conveniente incorporar uma função que permita, quando se carrega na tecla break, imprimir uma mensagem de paragem e voltar ao comando Basic.

Consideremos agora o problema de forçar a impressão de uma mensagem de erro. O Spectrum é capaz de imprimir mensagens de erro bastante claras, e não é de facto difícil utilizá-las e gerá-las nos nossos próprios programas apesar de não serem directamente aplicáveis a código-máquina. Forçar a impressão de uma mensagem deste modo não é particularmente difícil, em particular quando se sabe como, mas pode ser um tanto confuso nesta fase. Em vez disso, convirá usar um método que tenha a ver com a Basic e não com o código-máquina. Não se esqueça de que no caso de tentar imprimir o código dos primeiros 32 caracteres a máquina imprimirá uma mensagem de erro. A razão disto é que estes caracteres são usados pelo computador para instruções específicas e que podem ter a ver com a cor ou a introdução de uma linha. Como se sabe, aplica-se cor num programa carregando muito simplesmente em CAPS SHIFT e SYMBOL SHIFT e em seguida num dos números. Estamos então a introduzir no programa um desses códigos. Mas se escrevermos directamente o código equivalente, por exemplo PRINT CHR$ (13), esta operação resultará numa mensagem de erro. É fácil recorrer a este conhecimento ao construir um programa em código-máquina carregando no registo C um valor inferior a 32 e voltando à Basic.

O curto programa em código-máquina que se segue utiliza esta característica:


```
        KEY-P EQU 23557
        LASTK EQU 23560

START   LD A,(KEY-P)
        SUB 5
        JR NZ,START
BREAK?  LD A,(LAST K)
        SUB 127
        JR Z,EXIT
        LD C,(LAST K)
        LD B,0
        RET
EXIT    LD C,13
        LD B,0
        RET
```

O programa começa por etiquetar os dois endereços 23557 e 23560, utilizando como etiquetas os nomes das respectivas variáveis de sistema, KEY-P e LAST K. Deste modo, quando nos referimos a elas ao longo da rotina, podem-se usar nomes em vez do valor dos endereços. Isto torna mais fácil construir o programa e lê-lo mais tarde. A rotina começa por carregar o registo A com o valor guardado na variável de sistema KEY-P, subtraindo depois 5 a este valor (porque então no caso de a resposta ser zero terá sido premida uma tecla). De facto, como já sabemos, um valor de 5 naquele endereço indica que se carregou numa tecla.

Considerando que o resultado do registo A menos 5 é igual a zero, poderemos concluir que foi premida uma tecla; se esse valor não for zero a instrução de salto relativo para a flag zero igual a zero obrigará a execução a voltar a START e ler novamente o valor de KEY-P para o registo A.

Deste modo o programa entra num ciclo perpétuo até ser premida uma tecla, e como a variável de sistema KEY-P não nos diz qual é a tecla em causa teremos agora de verificar se é a tecla BREAK. Isto é feito quase do mesmo modo que usámos para verificar se se tinha premido uma tecla: carregamos no registo A o valor da variável de sistema LAST K, e subtraímos-lhe o código da tecla BREAK. Se esta for premida, o resultado será novamente zero, e a execução saltará para a rotina seguinte. Se não se carregou na tecla BREAK o valor correspondente à tecla premida é

carregado no registo C de modo a ser impresso no visor quando o comando voltar à Basic. É importante que asseguremos a existência de um valor zero no registo B porque, quando estivermos em Basic, não poderemos diferenciar entre os dois registos B e C e portanto qualquer número diferente de zero no registo B pode obviamente afectar o resultado.

Terminado isto, voltamos à Basic. A parte final do programa é formada pela curta rotina etiquetada EXIT, acedida apenas quando for premida a tecla BREAK, e que se limita a carregar no registo C um valor ilegal — ao mesmo tempo que carrega em B o valor zero. O computador devolve por si mesmo o comando à Basic. Como podemos ver, esta rotina pode ser guardada em qualquer zona da memória, podendo portanto ser gravada em cassette para uso futuro num programa Basic ou em código-máquina.

Tendo observado este processo, é fácil compreender porque razão não é normalmente possível sair de um programa em código-máquina, em particular no caso dos programas comerciais, dado que os autores raramente gostam que os outros vejam os seus programas. Usando este método pode-se no entanto evitar muitas dificuldades.

Passamos agora ao estudo de um dispositivo essencial de saída — o visor. À imagem impressa corresponde no interior do Spectrum um conjunto de informações guardadas de um modo bastante peculiar. Usam-se para tal 6 K bytes, contendo informações sobre os pontos ("pixels") que são ou não impressos. Além disso usa-se ainda cerca de 1 K para guardar as informações respeitantes à cor e outros atributos de cada caracter. Neste livro não estudaremos directamente esta questão dos atributos, e preocupar-nos-emos apenas com o problema da alta resolução e da produção de caracteres.

O visor é formado por linhas de 256 pontos ou pixels. Como cada um destes pode estar "ligado" ou "desligado", é possível guardar indicações sobre o estado de oito pixels em cada byte. Os oito pixels em causa são tratados como se formassem um número binário; se o pixel está "ligado" conta como um 1 binário, e no caso contrário como um 0 binário. O resultado é portanto um número binário de oito algarismo: 10101010.

Podemos demonstrar isto carregando no primeiro byte do ficheiro de imagem o valor binário acima indicado. Fazemo-lo

através da instrução POKE 16384, BIN 10101010 (Enter). Experimente em seguida carregar um arranjo diferente de algarismos binários, mas procure sempre não usar mais do que oito algarismos porque obterá então um valor decimal superior a 255 (valor máximo que o processador pode manipular). Este arranjo aplica-se a todo o visor — 6144 vezes. Podemos ilustrar o facto recorrendo ao curto programa seguinte:

```
10 FOR a = 16384 TO 16384 + 6143
20 POKE a, BIN 11111111
30 NEXT a
```

Experimente alterar o valor do número binário; poderá obter efeitos bastante interessantes. Notara que quando preenche a área onde se encontra o ficheiro de imagem com um dado algarismo, o visor não é preenchido sequencialmente, linha a linha. Em vez disso, o visor é dividido em três áreas diferentes. Cada uma destas é dividida por sua vez em oito blocos de caracteres. Primeiro é preenchida a primeira linha de cada um desses blocos, depois a segunda linha de cada um deles, etc. Ao terminar o preenchimento da última linha dos primeiros oito blocos o processo recomeça a partir do segundo terço do visor. Este sistema é de facto um tanto estranho...

Um outro modo de ilustrar o que se passa consiste em desenhar uma imagem no visor, gravando depois a imagem em fita (desde o endereço 16384 até 22527), carregando depois a imagem novamente. O ficheiro de imagem encontra-se sempre numa determinada área da memória, separada da área Basic ou de código-máquina, sendo por vezes carregada uma imagem no início de um programa como introdução atraente a este. Introduza o pequeno programa indicado a seguir, e grave-o em seguida em cassette usando a ordem SAVE "nome" CODE 16384,6144. Limpe o visor usando a ordem CLS, e carregue novamente o programa com LOAD ""CODE. O resultado deve ser bastante interessante.

```
10 LET n=631
20 LET a=120
25 PLOT 55,27: DRAW a,a,n*PI
```

Vejamos agora um outro exemplo. Carregue as duas rotinas

em código-máquina para deslocação de pixels para a esquerda e para a direita, escrevendo uma curta rotina para carregar a imagem anterior, deslocando-a em seguida para a esquerda; após uma rotação completa faça-a deslocar-se para a direita. Não é necessário escrever isto em código-máquina. O resultado ilustrará facilmente o modo de usar rotinas em código no interior de um programa Basic.

Como o leitor já compreendeu certamente, devido ao arranjo pouco habitual do visor em memória, a realização de uma tarefa bastante simples como a impressão de um caracter requer o uso de um programa relativamente complicado. No entanto, podemos concluir que sendo o próprio computador capaz de o fazer, a rotina necessária deverá encontrar-se em algum ponto da ROM e poderemos usar-nos desse facto. A importância desta rotina levou a colocá-la bem perto do início da memória, começando no endereço 16 (decimal) — mas como poderemos acedê-la? Felizmente foi concebida como módulo independente, o que permite chamá-la usando uma instrução CALL. Uma característica da concepção do processador Z80 é a possibilidade de aceder rapidamente algumas das rotinas mais usadas recorrendo a um conjunto de instruções que se aplicam apenas a oito dos primeiros endereços da memória; e o mais importante é que cada uma destas instruções ocupa apenas um byte! Uma destas instruções, precisamente a usada para aceder a rotina de impressão de uma imagem, é RST 16. Esta instrução tem o mesmo efeito que CALL 16, e termina por uma ordem RET que devolve o comando à primeira instrução que se segue a RST 16.

É necessário considerar ainda duas questões antes de fazer executar esta ordem. Em primeiro lugar é necessário definir a posição de PRINT, o que deverá ser feito em Basic porque em código-máquina, apesar de possível, é desnecessariamente complicado. Em segundo lugar, deve-se carregar no registo A o código do caracter a imprimir. Isto é útil não só devido à relativa facilidade de tradução de um caracter para o seu código, usando a tabela já fornecida, como ainda ao facto de permitir o uso de gráficos definidos pelo utilizador em código-máquina.

Vejamos agora uma rotina bastante simples para impressão do caracter "A" no canto superior esquerdo do visor. Não se esqueça de aceder ao código-máquina do modo indicado (em Basic):


```
10 PRINT AT 0,0;
20 RANDOMIZE USR 30000

ORG 30000
30000 3E61      LD a,97
30002 D7        RST 16
30003 C9        RET
```

Experimente agora alterar a posição de impressão e o conteúdo do registo A (o caracter a imprimir). Torna-se evidente a facilidade de imprimir um caracter gráfico definido por si carregando no registo A o código apropriado. É agora necessário um método que nos permita representar um caracter UDG em código-máquina. Infelizmente as rotinas usadas pelo computador para fazer isto em Basic não são acessíveis em código-máquina. Por outro lado, como se trata de qualquer modo de um processo complicado, vou fornecer ao leitor uma rotina capaz de permitir a formação de UDG's (gráficos definidos pelo utilizador). Não é verdadeiramente necessário que o leitor compreenda o modo como esta funciona — limite-se a usá-la. Introduza-a na máquina e aceda-a através de uma instrução CALL, seguida de 9 bytes de dados. O primeiro é o número do caracter gráfico, ou seja, a tecla A ou CHR$ 144 = 0, a tecla B ou CRH$ 145 = 1, etc. Torna-se assim possível especificar qual o caracter a definir. Os outros oito bytes são os "quadros de bits" que são usados da forma habitual, formando o novo caracter.

A primeira rotina que se segue define o caracter gráfico. A segunda é um exemplo em que se definem os caracteres 144 e 145 (teclas A e B).

```
org 30000
equ 65368 UDG
30000 0E 00        ld c,0
30002 E1           pop hl
30003 7E           ld a,(hl)
30004 A7           and a
30005 87           add a,a
30006 87           add a,a
30007 87           add a,a
30008 11 58 FF     ld de,UDG
```

```
30011 47          ld b,a
30012 7B          ld a,e
30013 80          add a,b
30014 5F          ld e,a
30015 7A          ld a,d
30016 06 00       ld b,0
30018 88          abd a,b
30019 57          ld d,a
Loop1
30020 79          ld a,c
30021 D6 08       sub 0
30023 30 07       jr nc,Exit
30025 0C          inc c
30026 23          inc hl
30027 7E          ld a,(hl)
30028 12          ld (de),a
30030 18 F4       jr Loop1
Exit
30032 23          inc hl
30033 E5          push hl
30034 C9          ret


org 28000
28000 CD 30 75    call C-Gen
defb 0
defb 1
defb 3
defb 7
defb 15
defb 31
defb 63
defb 127
defb 255
28012 CD 30 75    call C-Gen
defb 1
defb 255
defb 127
defb 63
defb 31
defb 15
```



```
defb 7
defb 3
defb 1

28024 C9          ret
```

Não é necessário que o leitor compreenda todas as ordens necessárias para escrever um programa de impressão de caracteres em código-máquina, mas como este programa pode ser-lhe muito útil decidimos apresentá-lo aqui:

```
9C40 64           LD H,H
9C41 64           LD H,H
9C42 3A409C       LD A,(9C40)
9C45 CB3F         SRL A
9C47 CB3F         SRL A
9C49 CB3F         SRL A
9C4B 4F           LD C,A
9C4C 3A419C       LD A,(9C41)
9C4F CB3F         SRL A
9C51 CB3F         SRL A
9C53 CB3F         SRL A
9C55 47           LD B,A
9C56 E600         AND 00
9C58 3E17         LD A,17
9C5A 98           SBC A,B
9C5B D8           RET C
9C5C 50           LD D,B
9C5D 47           LD B,A
9C5E 7A           LD A,D
9C5F CB27         SLA A
9C61 CB27         SLA A
9C63 CB27         SLA A
9C65 57           LD D,A
9C66 3A419C       LD A,(9C41)
9C69 92           SUB D
9C6A 57           LD D,A
9C6B F5           PUSH AF
9C6C 78           LD A,B
9C6D E618         AND 18
```

```
9C6F F640     OR 40
9C71 67       LD H,A
9C72 F1       POP AF
9C73 84       ADD A,H
9C74 67       LD H,A
9C75 78       LD A,B
9C76 E607     AND 07
9C78 0F       RRCA
9C79 0F       RRCA
9C7A 0F       RRCA
9C7B 81       ADD  A,C
9C7C 6F       LD L,A
9C7D 79       LD A,C
9C7E CB27     SLA A
9C80 CB27     SLA  A
9C82 CB27     SLA  A
9C84 47       LD B,A
9C85 3A409C   LD A,(9C40)
9C88 90       SUB B
9C89 FE00     CP 00
9C8B 281D     JR Z,9CAA
9C8D FE01     CP 01
9C8F 281C     JR Z,9CAD
9C91 FE02     CP 02
9C93 281B     JR Z ,9CB0
9C95 FE03     CP 03
9C97 281A     JR Z,9CB3
9C99 FE04     CP 04
9C9B 2819     JR Z,9CB6
9C9D FE05     CP 05
9C9F 2818     JR Z,9CB9
9CA1 FE06     CP 06
9CA3 2817     JR Z,9CBC
9CA5 FE07     CP 07
9CA7 2816     JR Z,9CBF
9CA9 C9       RET
9CAA CBFE     BIT 7,(HL)
9CAC C9       RET
9CAD CBF6     BIT 6,(HL)
9CAF C9       RET
```



```
9CB0 CBEE     BIT 5,(HL)
9CB2 C9       RET
9CB3 CBE6     BIT 4,(HL)
9CB5 C9       RET
9CB6 CBDE     BIT 3,(HL)
9CB8 C9       RET
9CB9 CBD6     BIT 2,(HL)
9CBB C9       RET
9CBC CBCE     BIT 1,(HL)
9CBE C9       RET
9CBF CBC6     BIT 0,(HL)
9CC1 C9       RET
9CC2 00       NOP
9CC3 00       NOP
9CC4 00       NOP
9CC5 00       NOP
```

GUARDAR E DEITAR FORA...

X

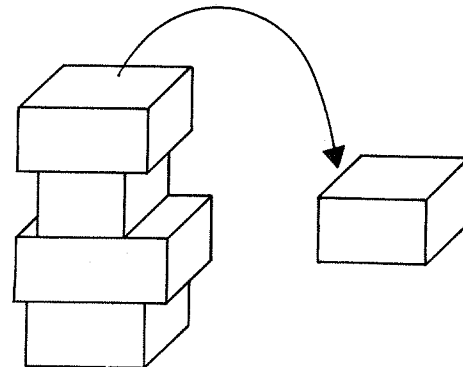
O "STACK"

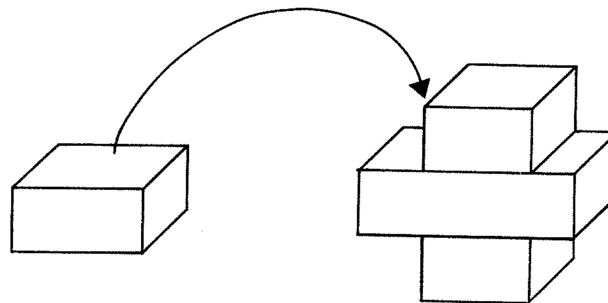
Este capítulo será dedicado à exploração de outras ordens em código-máquina, de alguns conceitos novos e de uma área particularmente útil usada pela própria CPU. Será necessário recordar as instruções INC e DEC, que nos conduzirão às instruções LDIR e LDDr — a compreensão destas depende da nossa familiaridade com as instruções INC e DEC.

Talvez pela primeira vez podemos agora tratar de um termo — em inglês, o "stack" — que significa precisamente o que é: uma "pilha" de números. Podemos em qualquer momento pôr ou tirar um número na parte superior do "stack", e apenas aí.

Pode-se pensar no "stack" como uma torre de caixas. É possível retirar a caixa que está por cima, ou acrescentar uma nova caixa; mas se mexermos nas que estão a meio a torre pode cair... É precisamente isto que existe em código-máquina — consistindo a única diferença no facto de cada caixa ser designada por um endereço. Note-se porém que uma das vantagens do "stack" consiste precisamente em não termos de nos preocupar com estes endereços.

Não existe qualquer restrição ao comprimento do "stack", se bem que quanto maior for menor será o espaço para tudo o resto. Construir o "stack" é bastante fácil; basta definir em que ponto da memória deve começar, e atribuir à variável de sistema STK-P esse valor. Cria-se deste modo um "indicador de stack" — o "stack pointer". Este indica constantemente a posição da memória onde será colocado o valor seguinte.





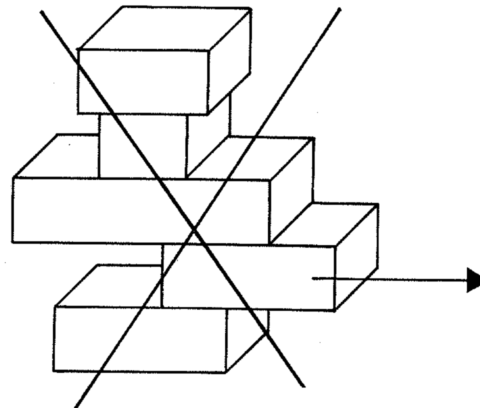


FIG. 35

```
org 30000
equ 28000 STK-P
30000 3E 10              ld a, 16
30002 FD 22 60 6D        ld (STK-P), 16
30006 21 60 6D           ld ht, STK-P
30009 35                 dec (hl)
```

Se não dispusermos de um Assembler este processo é um pouco mais difícil. Em primeiro lugar decide-se a posição onde se pretende guardar o Stack Pointer, depois verifica-se se o endereço por este indicado contém o valor zero, carrega-se o par de registos HL com este endereço, e carrega-se neste o valor do Stack Pointer.

```
30010 3E 10              ld a, 16
30012 21 60 6D           ld hl, 28000
30015 77                 ld (hl), a
30019 2B                 dec hl
```

Quando se coloca qualquer valor no Stack, o Stack Pointer diminui necessariamente de um, de modo a que da próxima vez que se acrescenta um valor este não se sobreponha ao anterior e seja de facto colocado no endereço imediatamente abaixo. Para enviar um valor para o Stack deve-se colocá-lo primeiramente no registo A. Este é então carregado no endereço guardado no Stack Pointer, sendo o valor deste diminuído de 1.

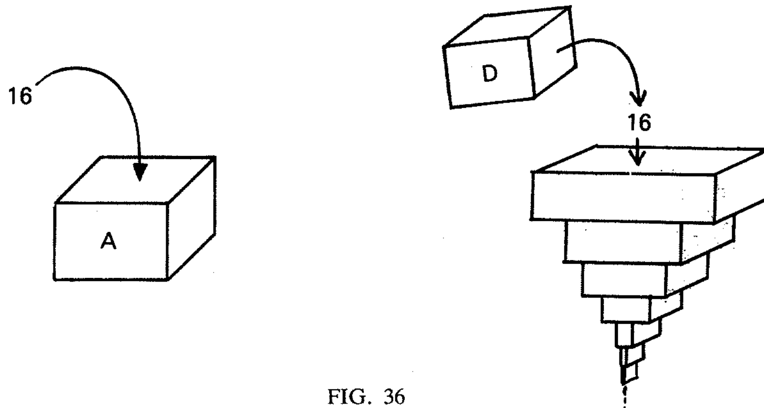


FIG. 36

Resta apenas desenvolver uma rotina que permita tirar um número do stack. Neste caso, carrega-se no registo A o valor guardado no topo do Stack, e aumenta-se em seguida de um o valor contido no Stack Pointer.

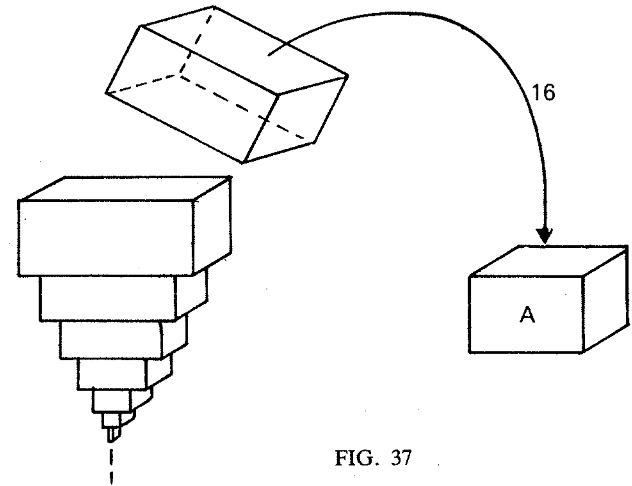


FIG. 37

Este programa permite-nos construir e usar um Stack simples. O uso de um Stack, ao qual é possível acrescentar ou retirar números, pode parecer uma perda de tempo, mas de facto pode-se revelar muito vantajoso constituindo um modo excelente de simplificar um programa. Provavelmente o modo mais simples de usar um Stack consiste em etiquetar as duas subrotinas usadas para pôr e tirar números do Stack com as palavras PUSH e POP, acedendo-as depois usando instruções CALL. Nestas condições copiam-se as instruções em código-máquina que fazem precisamente este trabalho mas que são mais complicadas de usar.

```
org 30000
equ 28000 STK-P
30000 3A 30 75           ld a, (STK-P)
30003 21 30 75           ld hl,STK-P
30006 34                 inc (hl)
```

```
30007 21 60 6D          ld hl, 28000
30010 7E                ld a, (hl)
30011 23                inc hl
```

Façamos agora um pequeno intervalo. O programa Basic que se segue cria um jogo baseado no mesmo princípio que o próprio Stack. Se bem que não seja escrito em código-máquina, a ideia do jogo tem directamente a ver com aquilo de que estamos a tratar neste capítulo, em particular com o uso do Stack. O princípio do jogo é muito antigo. A ideia consiste em transferir uma pilha de cinco anéis de diâmetros diferentes de um eixo para outro, devendo no entanto os diâmetros manter-se sempre por ordem decrescente. Só se pode mover um anel de cada vez, devendo ser sempre colocado sobre outro mais largo. Este jogo dará ao leitor uma ideia dos problemas que podem existir nos Stacks, em particular se não se sabe o que está por cima...

```
   2 GO SUB 2000
   5 BORDER 6: PAPER 6: INK 0: C L5
   7 DATA "1      "1        ","2        ","3          ","4          ","5
                 ","6
  10 RESTORE 7: LET C=10: LET M=0: DIM
A$(16,11): DIM C$(5,11): LET W=0: DIM B$(2,1): FOR
D=1 TO 16
  20 IF D<7 THEN READ A$(D): GO TO 40
  30 LET A$(D)=A$(6)
  40 NEXT 0
 400 FOR X=1 TO 5: LET C$(X)=A$(X): NEXT X
 470 PRINT AT 0,8; "TORRES"
 475 PRINT AT 20, 18; M;" Movimentos"
 480 FOR D=0 TO 31. PRINT AT 16, D; "  "; NEXT D:
PRINT AT 17,4; INK 1; "1"; TAB 14; "2"; TAB 24; "3"
 500 FOR B=1 TO 15: LET C=C+1: PRINT AT C,INT
(B/5-.1)*10; A$(B,2 TO )
 510 IF C=15 THEN LET C=10
 520 NEXT B
 530 IF W=1 THEN GO TO 1600
 580 INPUT "Do eixo?"; LINE B$(1), "Para eixo?";
LINE B$(2): PRINT AT 21,15; "                    ": IF
B$(1) <STR$ 1 OR B$(1)>STR$ 3 OR B$(2) <STR$ 1 OR
B$(2)>STR$ 3 THEN GO TO 1500
 600 FOR Z=1 TO 5
 610 IF A$ ( (VAL B$(1) -1) *5+Z) <>A$(16) THEN GO
TO 640
 615 IF Z=5 AND A$ ( (VAL B$(1) -1) *5+Z)=A$(16)
THEN GO TO 1500
 620 NEXT Z
 640 FOR Y=5 TO 1 STEP -1
 660 IF A$ ( (VAL B$(2) -1) *5+Y)=A$(16) THEN GO
TO 1000
 680 NEXT Y
1000 IF Y=5 THEN GO TO 1010
1003 IF A$ ( (VAL B$(2) -1) *5+Y+1) <A$ ( (VAL B$(1)
-1) *5+Z) THEN GO TO 1500
1010 LET A$ ( (VAL B$(2) -1) *5+Y)=R$ ( (VAL B$(1)
-1) *5+Z)
1020 LET A$ ( (VAL B$(1) -1) *5+Z)=A$(16)
1030 LET M=M+1
1040 FOR D=1 TO 5: IF A$(D+10) <> C$(D) THEN GO
TO 470
1050 NEXT D: LET W=1: GO TO 470
1500 BEEP 1, -20: PRINT AT 21,15; FLASH 1; "ILE
GAL"
1520 GO TO 580
1600 PRINT AT 5,11; FLASH 1; "ÓPTIMO"; FLASH
0;      TAB 5; "Conseguiu em "; M; " Movimentos";
FLASH 0: FOR x=0 TO 255: OUT 254,X: NEXT X: GO TO
1605
1601 PRINT TAB 5; "Mas pode fazer melhor!"
1610 INPUT "OUTRO JOGO (s/n)?"; Q$: IF Q$<> "s"
THEN PRINT AT 21,0; "Adeus..."; STOP
1620 RUN
2000 RESTORE 3000
2005 FOR y=0 TO 5
2010 FOR x=0 TO 7
2020 READ chr: POKE 65358+y*8+x,chr
2030 NEXT x
2040 NEXT y
2050 RETURN
```

```
3000 DATA 0,0,BIN 00111111,BIN 01111111,BIN
01111111,BIN 00111111,0,0
3010 DATA 0,0,255,255,255,255,0,0
3020 DATA 0,0,BIN 11111100,BIN 11111110,BIN
11111110,BIN 11111100,0,0
3030 DATA 60,60,60,60,60,60,60,60
3040 DATA 0,24,60,60,60,60,60,60
3050 DATA 0,0,BIN 01111110,255,255,BIN 01111110,0,0
```

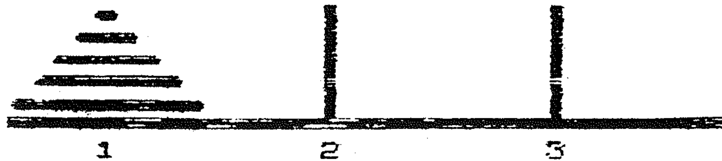


FIG. 38

A linguagem Basic, directa e indirectamente, usa três Stacks diferentes: o Stack de GO SUB, que guarda os endereços de retorno das subrotinas; o Stack Calculador, usado para todas as operações numéricas; e finalmente o Stack-máquina, usado pela própria CPU, acessível pelo código-máquina. Não é fácil localizar ou usar este último em Basic, sendo sempre aconselhável passar a código-máquina antes de o utilizar.

Observámos anteriormente as instruções INC e DEC e o modo como estas levam o conteúdo de um registo ou endereço a ser INCrementado ou DECrementado. Existem certas utilizações em que a vantagem destas instruções é óbvia. Um exemplo é a transferência do conteúdo de um bloco de memória para outro. Neste caso usamos dois indicadores para referenciar as duas áreas que podem, por uma questão de conveniência, ser designadas Fonte e Destino. As ordens INC e DEC são depois usadas para aumentar ou diminuir de um os valores contidos nesses indicadores, evitando a sua alteração forçada antes de cada transferência. Por exemplo:


```
ld hl,fonte
ld de,destino
ld a,(hl)
ld (de),a
inc hl

inc de
ld a,(hl)
ld (de),a
inc hl
inc de
,,
,,
```

Isto pode ser satisfatório para transferir um ou dois bytes, mas para um trabalho mais extenso torna-se bastante trabalhoso. Felizmente podemos recorrer a um ciclo, mas antes de usar esta ordem devemos especificar, num dado registo, o número de vezes que o ciclo deve ser executado. Isto é feito do seguinte modo:

```
ld hl,fonte
ld de,destino
ld bc,nº de vezes
loop ld a,(hl)
ld (de),a
inc de
inc hl
```

```
dec bc
ld a,b
add a,c
jr nz,loop
ret
```

Somando o conteúdo do byte mais significativo (B) ao do byte menos significativo (C) é possível verificar se o conteúdo do par de registos é zero. Isto fornece-nos portanto um meio relativamente simples de transferir um bloco de memória.

A rotina anterior pode ser ainda mais simplificada usando duas ordens que requerem apenas dois bytes cada (INC e DEC requerem apenas um byte cada), tornando possível eliminar todas as ordens da rotina menos quatro. Estas duas novas ordens são "LDIR" e "LDDR", e as suas definições são:

LDIR = "Transferência condicional em memória com auto-incremento dos registos indicadores, e auto-decremento de um registo contador".

LDDR = "Transferência condicional em memória com auto-decremento dos registos indicadores e auto-decremento de um registo contador".

Tudo isto significa apenas que é possível, usando a ordem LDIR, carregar o par de registos HL o início do código-fonte; no par de registos DE o início da área de destino; e no par de registos BC o número de bytes a transferir. Em uso, o número de bytes especificado em BC é transferido da posição HL para a posição DE, sendo o conteúdo destes dois registos incrementado ao terminar cada transferência. O valor em BC é por sua vez decrementado até atingir zero, terminando nesse momento a execução da ordem e passando-se à instrução seguinte. A ordem LDDR funciona exactamente da mesma forma, exceptuando evidentemente o facto de o conteúdo dos dois indicadores ser diminuído em vez de aumentado. Uma rotina que usasse estas ordens teria a seguinte aparência:


```
LD HL, fonte
LD DE, destino
LD BC, bytes
LDIR
RET
```

Por exemplo, uma rotina que transferisse todo o conteúdo do visor teria a seguinte aparência:

```
org 25501
25501 21 00 40      ld hl,16354
25504 11 00 64      ld de,25600
25507 01 00 1C      ld bc,7168
25510 ED B0         ldir
25512 C9            ret

25513 21 00 64      ld hl,25600
25516 11 00 40      ld de,16364
25519 01 00 1C      ld bc,7168
25522 ED B0         ldir
25524 C9            ret
```

Escreva LOAD no programa de desenho Basic apresentado anteriormente, guarde uma imagem usando a primeira das duas rotinas anteriores, limpe o visor (CLS) e transfira a imagem novamente usando a segunda rotina. Note a incrível velocidade de execução desta ordem.

Chegamos assim ao final da nossa viagem introdutória pelos aspectos essenciais da programação em código-máquina — uma viagem que espero tenha sido agradável e esclarecedora. Se desenvolveu no leitor um desejo de aprofundar o assunto, terá valido a pena. Se o equipou com instrumentos suficientes para utilizar a informação contida em textos mais avançados, terei conseguido o objectivo essencial.

Boa programação

J. M. C. W. Abril 1983

APÊNDICE A

00	0	00000000
01	1	00000001
02	2	00000010
03	3	00000011
04	4	00000100
05	5	00000101
06	6	00000110
07	7	00000111
08	8	00001000
09	9	00001001
0A	10	00001010
0B	11	00001011
0C	12	00001100
0D	13	00001101
0E	14	00001110
0F	15	00001111
10	16	00010000
11	17	00010001
12	18	00010010
13	19	00010011
14	20	10010100
15	21	10010101
16	22	00010110
17	23	00010111
18	24	00011000
19	25	00011001
1A	26	00011010
1B	27	00011011

1C	28	00011100
1D	29	00011101
1E	30	00011110
1F	31	00011111
20	32	00100000
21	33	00100001
22	34	00100010
23	35	00100011
24	36	00100100
25	37	00100101
26	38	00100110
27	39	00100111
28	40	00101000
29	41	00101001
2A	42	00101010
2B	43	00101011
2C	44	00101100
2D	45	00101101
2E	46	00101110
2F	47	00101111
30	48	00110000
31	49	00110001
32	50	00110010
33	51	00110011
34	52	00110100
35	53	00110101
36	54	00110110
37	55	00110111
38	56	00111000
39	57	00111001
3A	58	00111010
3B	59	00111011
3C	60	00111100
3D	61	00111101
3E	62	00111110
3F	63	00111111
40	64	01000000
41	65	01000001
42	66	01000010
43	67	01000011
44	68	01000100

45	69	01000101
46	70	01000110
47	71	01000111
48	72	01001000
49	73	01001001
4A	74	01001010
4B	75	01001011
4C	76	01001100
4D	77	01001101
4E	78	01001110
4F	79	01001111
50	80	01010000
51	81	01010001
52	82	01010010
53	83	01010011
54	84	01010100
55	85	01010101
56	86	01010110
57	87	01010111
58	88	01011000
59	89	01011001
5A	90	01011010
5B	91	01011011
5C	92	01011100
5D	93	01011101
5E	94	01011110
5F	95	01011111
60	96	01100000
61	97	01100001
62	98	01100010
63	99	01100011
64	100	01100100
65	101	01100101
66	102	01100110
67	103	01100111
68	104	01101000
69	105	01101001
6A	106	01101010
6B	107	01101011
6C	108	01101100

6D	109	01101101
6E	110	01101110
6F	111	01101111
70	112	01110000
71	113	01110001
72	114	01110010
73	115	01110011
74	116	01110100
75	117	01110101
76	118	01110110
77	119	01110111
78	120	01111000
79	121	01111001
7A	122	01111010
7B	123	01111011
7C	124	01111100
7D	125	01111101
7E	126	01111110
7F	127	01111111
80	128	10000000
81	129	10000001
82	130	10000010
83	131	10000011
84	132	10000100
85	133	10000101
86	134	10000110
87	135	10000111
88	136	10001000
89	137	10001001
8A	138	10001010
8B	139	10001011
8C	140	10001100
8D	141	10001101
8E	142	10001110
8F	143	10001111
90	144	10010000
91	145	10010001
92	146	10010010
93	147	10010011
94	148	10010100
95	149	10010101

96	150	10010110
97	151	10010111
98	152	10011000
99	153	10011001
9A	154	10011010
9B	155	10011011
9C	156	10011100
9D	157	10011101
9E	158	10011110
9F	159	10011111
A0	160	10100000
A1	161	10100001
A2	162	10100010
A3	163	10100011
A4	164	10100100
A5	165	10100101
A6	166	10100110
A7	167	10100111
A8	168	10101000
A9	169	10101001
AA	170	10101010
AB	171	10101011
AC	172	10101100
AD	173	10101101
AE	174	10101110
AF	175	10101111
B0	176	10110000
B1	177	10110001
B2	178	10110010
B3	179	10110011
B4	180	10110100
B5	181	10110101
B6	182	10110110
B7	183	10110111
B8	184	10111000
B9	185	10111001
BA	186	10111010
BB	187	10111011
BC	188	10111100
BD	189	10111101
BE	190	10111110

BF	191	10111111
C0	192	11000000
C1	193	11000001
C2	194	11000010
C3	195	11000011
C4	196	11000100
C5	197	11000101
C6	198	11000110
C7	199	11000111
C8	200	11001000
C9	201	11001001
CA	202	11001010
CB	203	11001011
CC	204	11001100
CD	205	11001101
CE	206	11001110
CF	207	11001111
D0	208	11010000
D1	209	11010001
D2	210	11010010
D3	211	11010011
D4	212	11010100
D5	213	11010101
D6	214	11010110
D7	215	11010111
D8	216	11011000
D9	217	11011001
DA	218	11011010
DB	219	11011011
DC	220	11011100
DD	221	11011101
DE	222	11011110
DF	223	11011111
E0	224	11100000
E1	225	11100001
E2	226	11100010
E3	227	11100011
E4	228	11100100
E5	229	11100101
E6	230	11100110
E7	231	11100111
E8	232	11101000
E9	233	11101001
EA	234	11101010
EB	235	11101011
EC	236	11101100
ED	237	11101101
EE	238	11101110
EF	239	11101111
F0	240	11110000
F1	241	11110001
F2	242	11110010
F3	243	11110011
F4	244	11110100
F5	245	11110101
F6	246	11110110
F7	247	11110111
F8	248	11111000
F9	249	11111001
FA	250	11111010
FB	251	11111011
FC	252	11111100
FD	253	11111101
FE	254	11111110
FF	255	11111111

```
 10 FOR a = 0 TO 255
 20 GO SUB 400: LPRINT "        "; a; "        "; GO SUB
200: LPRINT
 30 NEXT a
200 LET n = a
210 FOR x = 7 TO 0 STEP -1
220 IF INT (n/ (2↑x)) = 1 THEN GO TO 300
230 LPRINT 0;
240 NEXT x
250 RETURN
300 LPRINT 1;
310 LET n = n -2± x
320 GO TO 240
```

```
400 LET n = a
410 LET x = INT (n/16)
420 LET y = ( (n/16) -INT (n/16) )*16
430 IF x>9 THEN LET x = x + 7
440 IF y>g THEB LET y = y + 7
450 LET x = x + 48: LET y = y + 48
460 LPRINT CHR$ x; CHR$ y;
470 RETURN
```


APÊNDICE B

Códigos de operação Z80 ordenados por MNEMÓNICAS

Op code		Hex	Decimal
ADC	A,(HL)	8E	142
ADC	A,(IX+d)	DD8Edd	221,142,XX
ADC	A,(IY+d)	FD8Edd	253,142,XX
ADC	A,A	8F	143
ADC	A,B	88	136
ADC	A,C	89	137
ADC	A,D	8A	138
ADC	A,E	8B	139
ADC	A,H	8C	140
ADC	A,L	8D	141
ADC	A,xx	CExx	206,XX
ADC	HL,BC	ED4A	237,74
ADC	HL,DE	ED5A	237,90
ADC	HL,HL	ED6A	237,106
ADC	HL,SP	ED7A	237,122
ADD	A,(HL)	86	134
ADD	A,(IX+d)	DD86dd	221,134,XX
ADD	A,(IY+d)	FD86dd	253,134,XX
ADD	A,A	87	135
ADD	A,B	80	128
ADD	A,C	81	129
ADD	A,D	82	130
ADD	A,E	83	131
ADD	A,H	84	132
ADD	A,L	85	133
ADD	A,xx	C6xx	198,XX

ADD	HL,BC	09	9
ADD	HL,DE	19	25
ADD	HL,HL	29	41
ADD	HL,SP	39	57
ADD	IX,BC	DD09	221,9
ADD	IX,DE	DD19	221,25
ADD	IX,IX	DD28	221,41
ADD	IX,SP	DD39	221,57
ADD	IY,BC	FD09	253,9
ADD	IY,DE	FD19	253,25
ADD	IY,IY	FD29	253,41
ADD	IY,SP	FD39	253,57
AND	(HL)	A6	166
AND	(IX+d)	DDA6dd	221,166,XX
AND	(IY+d)	FDA6dd	253,166,XX
AND	A	A7	167
AND	B	A0	160
AND	C	A1	161
AND	D	A2	162
AND	E	A3	163
AND	H	A4	164
AND	L	A5	165
AND	xx	E6xx	230,XX
BIT	0,(HL)	CB46	203,70
BIT	0,(IX+d)	DDCBdd46	221,203,XX,70
BIT	0,(IY+d)	FDCBdd46	253,203,XX,70
BIT	0,A	CB47	203,71
BIT	0,B	CB40	203,64
BIT	0,C	CB41	203,65
BIT	0,D	CB42	203,66
BIT	0,E	CB43	203,67
BIT	0,H	CB44	203,68
BIT	0,L	CB45	203,69
BIT	1,(HL)	CB4E	203,78
BIT	1,(IX+d)	DDCBdd4E	221,203,XX,78
BIT	1,(IY+d)	FDCBdd4E	253,203,XX,78
BIT	1,A	CB4F	203,79
BIT	1,B	CB48	203,72
BIT	1,C	CB49	203,73
BIT	1,D	CB4A	203,74
BIT	1,E	CB4B	203,75
BIT	1,H	CB4C	203,76
BIT	1,L	CB4D	103,77
BIT	2,(HL)	CB56	203,86
BIT	2,(IX+d)	DDCBdd56	221,203,XX,86

BIT	2,(IY+d)	FDCBdd56	253,203,XX,86
BIT	2,A	CB57	203,87
BIT	2,B	CB50	203,80
BIT	2,C	CB51	203,81
BIT	2,D	CB52	203,82
BIT	2,E	CB53	203,83
BIT	2,H	CB54	203,84
BIT	2,L	CB55	203,85
BIT	3,(HL)	CB5E	203,94
BIT	3,(IX+d)	DDCBdd5E	221,203,XX,94
BIT	3,(IY+d)	FDCBdd5E	253,203,XX,94
BIT	3,A	CB5F	203,95
BIT	3,B	CB58	203,88
BIT	3,C	CB59	203,89
BIT	3,D	CB5A	203,90
BIT	3,E	CB5B	203,91
BIT	3,H	CB5C	203,92
BIT	3,L	CB5D	203,93
BIT	4,(HL)	CB66	203,102
BIT	4,(IX+d)	DDCBdd66	221,203,XX,102
BIT	4,(IY+d)	FDCBdd66	253,203,XX,102
BIT	4,A	CB67	203,103
BIT	4,B	CB60	203,96
BIT	4,C	CB61	203,97
BIT	4,D	CB62	203,98
BIT	4,E	CB63	203,99
BIT	4,H	CB64	203,100
BIT	4,L	CB65	203,101
BIT	5,(HL)	CB6E	203,110
BIT	5,(IX+d)	DDCBdd6E	221,203,XX,110
BIT	5,(IY+d)	FDCBdd6E	253,203,XX,110
BIT	5,A	CB6F	203,111
BIT	5,B	CB68	203,104
BIT	5,C	CB69	203,105
BIT	5,D	CB6A	203,106
BIT	5,E	CB6B	203,107
BIT	5,H	CB6C	203,108
BIT	5,L	CB6D	203,109
BIT	6,(HL)	CB76	203,118
BIT	6,(IX+d)	DDCBdd76	221,203,XX,118
BIT	6,(IY+d)	FDCBdd76	253,203,XX,118
BIT	6,A	CB77	203,119
BIT	6,B	CB70	203,112
BIT	6,C	CB71	203,113
BIT	6,D	CB72	203,114

BIT	6,E	CB73	203,115
BIT	6,H	CB74	203,116
BIT	6,L	CB75	203,117
BIT	7,(HL)	CB7E	203,116
BIT	7,(IX+d)	DDCBdd7E	221,203,XX,126
BIT	7,(IY+d)	FDCBdd7E	253,203,XX,126
BIT	7,A	CB7F	203,127
BIT	7,B	CB78	203,120
BIT	7,C	CB69	203,121
BIT	7,D	CB7A	203,122
BIT	7,E	CB7B	203,123
BIT	7,H	CB7C	203,124
BIT	7,L	CB7D	203,125
CALL	C,xxxx	DCxxxx	220,XX,XX
CALL	M,xxxx	FCxxxx	252,XX,XX
CALL	NC,xxxx	D4xxxx	212,XX,XX
CALL	NZ,xxxx	C4xxxx	196,XX,XX
CALL	P,xxxx	F4xxxx	244,XX,XX
CALL	PE,xxxx	ECxxxx	236,XX,XX
CALL	PO,xxxx	E4xxxx	228,XX,XX
CALL	xxxx	CDxxxx	205,XX,XX
CALL	Z,xxxx	CCxxxx	204,XX,XX
CCF		3F	63
CP	(HL)	BE	190
CP	(IX+d)	DDBEdd	221,190,XX
CP	(IY+d)	FDBEdd	253,190,XX
CP	A	BF	191
CP	B	B8	184
CP	C	B9	185
CP	D	BA	186
CP	E	BB	187
CP	H	BC	188
CP	L	BD	189
CP	xx	FExx	254,XX
CPD		EDA9	237,169
CPDR		EDB9	237,185
CPI		EDA1	237,161
CPIR		EDB1	237,177
CPL		2F	47
DAA		27	39
DEC	(HL)	35	53
DEC	(IX+d)	DD35dd	221,53,XX
DEC	(IY+d)	FC35dd	253,53,XX
DEC	A	3D	61
DEC	B	05	5

DEC	BC	0B	11
DEC	C	0D	13
DEC	D	15	21
DEC	DE	1B	27
DEC	E	1D	29
DEC	H	25	37
DEC	HL	2B	43
DEC	IX	DD2B	221,43
DEC	IY	FD2B	253,43
DEC	L	2D	45
DEC	SP	3B	59
DI		F3	243
DJNZ	xx	10xx	16,XX
EI		FB	251
EX	(SP),HL	E3	227
EX	(SP),IX	DDE3	221,227
EX	(SP),IY	FDE3	253,227
EX	AF,AF	08	8
EX	DE,HL	EB	235
Exx		D9	217
HALT		76	118
IM	0	ED46	237,70
IM	1	ED56	237,86
IM	2	ED5E	237,94
IN	A,(C)	ED78	237,120
IN	A,(xx)	DBxx	219,XX
IN	B,(C)	ED40	237,64
IN	C,(C)	ED48	237,72
IN	D,(C)	ED50	237,80
IN	E,(C)	ED58	237,88
IN	H,(C)	ED60	237,96
IN	L,(C)	ED68	237,104
INC	(HL)	34	52
INC	(IX+d)	DD34dd	221,52,XX
INC	(IY+d)	FC34dd	253,52,XX
INC	A	3C	60
INC	B	04	4
INC	BC	03	3
INC	C	0C	12
INC	D	14	20
INC	DE	13	19
INC	E	1C	28
INC	H	24	36
INC	HL	23	35
INC	IX	DD23	221,35

INC	IY	FD23	253,35
INC	L	2C	44
INC	SP	33	51
IND		EDAA	237,170
INDR		EDBA	237,186
INI		EDA2	237,162
INIR		EDB2	237,178
JP	(HL)	E9	233
JP	(IX)	DDE9	221,233
JP	(IY)	FDE9	253,233
JP	C,xxxx	DAxxxx	218,XX,XX
JP	M,xxxx	FAxxxx	250,XX,XX
JP	NC,xxxx	D2xxxx	210,XX,XX
JP	NZ,xxxx	C2xxxx	194,XX,XX
JP	P,xxxx	F2xxxx	242,XX,XX
JP	PE,xxxx	EAxxxx	234,XX,XX
JP	PO,xxxx	E2xxxx	234,XX,XX
JP	xxxx	C3xxxx	195,XX,XX
JP	Z,xxxx	CAxxxx	202,XX,XX
JR	C,xx	38xx	56,XX
JR	NC,xx	30xx	48,XX
JR	NZ,xx	20xx	32,XX
JR	xx	18xx	24,XX
JR	Z,xx	28xx	40,XX
LD	(BC),A	02	2
LD	(DE),A	12	18
LD	HL,(xxxx)	2Axxxx	42,XX,XX
LD	(HL),A	77	119
LD	(HL),B	70	112
LD	(HL),C	71	113
LD	(HL),D	72	114
LD	(HL),E	73	115
LD	(HL),H	74	116
LD	(HL),L	75	117
LD	(HL),xx	36xx	54,XX
LD	(IX+d),A	DD77dd	221,119,XX
LD	(IX+d),B	DD70dd	221,112,XX
LD	(IX+d),C	DD71dd	221,113,XX
LD	(IX+d),D	DD72dd	221,114,XX
LD	(IX+d),E	DD73dd	221,115,XX
LD	(IX+d),H	DD74dd	221,116,XX
LD	(IX+d),L	DD75dd	221,117,XX
LD	(IX+d),xx	DD36ddxx	221,54,XX,XX
LD	(IY+d),A	FD77dd	253,119,XX
LD	(IY+d),B	FD70dd	253,112,XX

LD	(IY+d),C	FD71dd	253,113,XX
LD	(IY+d),D	FD72dd	253,114,XX
LD	(IY+d),E	FD73dd	253,115,XX
LD	(IY+d),H	FD74dd	253,116,XX
LD	(IY+d),L	FD75dd	253,117,XX
LD	(IY+d),xx	D36ddxx	253,54,XX,XX
LD	(xxxx),A	32xxxx	50,XX,XX
LD	(xxxx),BC	ED43xxxx	237,67,XX,XX
LD	(xxxx),DE	ED53xxxx	237,83,XX,XX
LD	(xxxx),HL	22xxxx	34,XX,XX
LD	(xxxx),IX	DD22xxxx	221,34,XX,XX
LD	(xxxx),IY	FD22xxxx	253,34,XX,XX
LD	(xxxx),SP	ED73xxxx	237,115,XX,XX
LD	A,(BC)	0A	10
LD	A,(DE)	1A	26
LD	A,(HL)	7E	126
LD	A,(IX+d)	DD7Edd	221,126,XX
LD	A,(IY+d)	FD7Edd	253,126,XX
LD	A,(xxxx)	3Axxxx	58,XX,XX
LD	A,A	7F	127
LD	A,B	78	120
LD	A,C	79	121
LD	A,D	7A	122
LD	A,E	7B	123
LD	A,H	7C	124
LD	A,I	ED57	237,87
LD	A,L	7D	125
LD	A,R	ED5F	237,85
LD	A,xx	3Exx	62,XX
LD	B,(HL)	46	70
LD	B,(IX+d)	DD46dd	221,70,XX
LD	B,(IY+d)	FD46dd	253,70,XX
LD	B,A	47	71
LD	B,B	40	64
LD	B,C	41	65
LD	B,D	42	66
LD	B,E	43	67
LD	B,H	44	68
LD	B,L	45	69
LD	B,xx	06xx	6,XX
LD	BC,(xxxx)	ED4Bxxxx	237,75,XX,XX
LD	BC,xxxx	01xxxx	1,XX,XX
LD	C,(HL)	4E	78
LD	C,(IX+d)	DD4Edd	221,78,XX
LD	C,(IY+d)	FD4Edd	253,78,XX

LD	C,A	4F	79
LD	C,B	48	72
LD	C,C	49	73
LD	C,D	4A	74
LD	C,E	4B	75
LD	C,H	4C	76
LD	C,L	4D	77
LD	C,xx	0Exx	14,XX
LD	D,(HL)	56	86
LD	D,(IX+d)	DD56dd	221,86,XX
LD	D,(IY+d)	FD56dd	253,86,XX
LD	D,A	57	87
LD	D,B	50	80
LD	D,C	51	81
LD	D,D	52	82
LD	D,E	53	83
LD	D,H	54	84
LD	D,L	55	85
LD	D,xx	16xx	22,XX
LD	DE,(xxxx)	ED5Bxxxx	237,91,XX,XX
LD	DE,xxxx	11xxxx	17,XX,XX
LD	E,(HL)	5E	94
LD	E,(IX+d)	DD5Edd	221,94,XX
LD	E,(IY+d)	FD5Edd	253,94,XX
LD	E,A	5F	95
LD	E,B	58	88
LD	E,C	59	89
LD	E,D	5A	90
LD	E,E	5B	91
LD	E,H	5C	92
LD	E,L	5D	93
LD	E,xx	1Exx	30,XX
LD	H,(HL)	66	102
LD	H,(IX+d)	DD66dd	221,102,XX
LD	H,(IY+d)	FD66dd	253,102,XX
LD	H,A	67	103
LD	H,B	60	96
LD	H,C	61	97
LD	H,D	62	98
LD	H,E	63	99
LD	H,H	64	100
LD	H,L	65	101
LD	H,xx	26xx	38,XX
LD	HL,xxxx	21xxxx	33,XX,XX
LD	I,A	ED47	237,71

LD	IX,xxxx	DD21xxxx	221,33,XX,XX
LD	IY,xxxx	FD2Axxxx	253,42,XX,XX
LD	IY,xxxx	FD21xxxx	253,33,XX,XX
LD	L,(HL)	6E	110
LD	L,(IX+d)	DD6Edd	221,110,XX
LD	L,(IY+d)	FD6Edd	253,110,XX
LD	L,A	6F	111
LD	L,B	68	104
LD	L,C	69	105
LD	L,D	6A	106
LD	L,E	6B	107
LD	L,H	6C	108
LD	L,L	6D	109
LD	L,xx	2Exx	46,XX
LD	R,A	ED4F	237,79
LD	SP,(xxxx)	ED7Bxxxx	237,123,XX,XX
LD	SP,HL	F9	249
LD	SP,IX	DDF9	221,249
LD	SP,IY	FDF9	253,249
LD	SP,xxxx	31xxxx	49,XX,XX
LDD		EDA8	237,168
LDDR		EDB8	237,184
LDI		EDA0	237,160
LDIR		EDB0	237,176
NEG		ED44	237,68
NOP		00	0
OR	(HL)	B6	182
OR	(IX+d)	DDB6dd	221,182,XX
OR	(IY+d)	FDB6dd	253,182,XX
OR	A	B7	183
OR	B	B0	176
OR	C	B1	177
OR	D	B2	178
OR	E	B3	179
OR	H	B4	180
OR	L	B5	181
OR	xx	F6xx	246,XX
OTDR		EDBB	237,187
OTIR		EDB3	237,179
OUT	(C),A	ED79	237,121
OUT	(C),B	ED41	237,65
OUT	(C),C	ED49	237,73
OUT	(C),D	ED51	237,81
OUT	(C),E	ED59	237,89
OUT	(C),H	ED61	237,97

OUT	(C),L	ED69	237,105
OUT	(xx),A	D3xx	211,XX
OUTD		EDAB	237,171
OUTI		EDA3	237,163
POP	AF	F1	241
POP	BC	C1	193
POP	DE	D1	209
POP	HL	E1	225
POP	IX	DDE1	221,225
POP	IY	FDE1	253,225
PUSH	AF	F5	245
PUSH	BC	C5	197
PUSH	DE	D5	213
PUSH	HL	E5	229
PUSH	IX	DDE5	221,229
PUSH	IY	FDE5	253,229
RES	0,(HL)	CB86	203,134
RES	0,(IX+d)	DDCBdd86	221,203,XX,134
RES	0,(IY+d)	FDCBdd86	253,203,XX,134
RES	0,A	CB87	203,135
RES	0,B	CB80	203,128
RES	0,C	CB81	203,129
RES	0,D	CB82	203,130
RES	0,E	CB83	203,131
RES	0,H	CB84	203,132
RES	0,L	CB85	203,133
RES	1,(HL)	CB8E	203,142
RES	1,(IX+d)	DDCBdd8E	221,203,XX,142
RES	1,(IY+d)	FDCBdd8E	253,203,XX,142
RES	1,A	CB8F	203,143
RES	1,B	CB88	203,136
RES	1,C	CB89	203,137
RES	1,D	CB8A	203,138
RES	1,E	CB8B	203,139
RES	1,H	CB8C	203,140
RES	1,L	CB8D	203,141
RES	2,(HL)	CB96	203,150
RES	2,(IX+d)	DDCBdd96	221,203,XX,150
RES	2,(IY+d)	FDCBdd96	253,203,XX,150
RES	2,A	CB97	203,151
RES	2,B	CB90	203,144
RES	2,C	CB91	203,145
RES	2,D	CB92	203,146
RES	2,E	CB93	203,147
RES	2,H	CB94	203,148

RES	2,L	CB95	203,149
RES	3,(HL)	CB9E	203,158
RES	3,(IX+d)	DDCBdd9E	221,203,XX,158
RES	3,(IY+d)	FDCBdd9E	253,203,XX,158
RES	3,A	CB9F	203,159
RES	3,B	CB98	203,152
RES	3,C	CB99	203,153
RES	3,D	CB9A	203,154
RES	3,E	CB9B	203,155
RES	3,H	CB9C	203,156
RES	3,L	CB9D	203,157
RES	4,(HL)	CBA6	203,166
RES	4,(IX+d)	DDCBddA6	221,203,XX,166
RES	4,(IY+d)	FDCBddA7	253,203,XX,166
RES	4,A	CBA7	203,167
RES	4,B	CBA0	203,160
RES	4,C	CBA1	203,161
RES	4,D	CBA2	203,162
RES	4,E	CBA3	203,163
RES	4,H	CBA4	203,164
RES	4,L	CBA5	203,165
RES	5,(HL)	CBAE	203,174
RES	5,(IX+d)	DDCBddAE	221,103,XX,174
RES	5,(IY+d)	FDCBddAE	253,203,XX,174
RES	5,A	CBAF	203,175
RES	5,B	CBA8	203,168
RES	5,C	CBA9	203,169
RES	5,D	CBAA	203,170
RES	5,E	CBAB	203,171
RES	5,H	CBAC	203,172
RES	5,L	CBAD	203,173
RES	6,(HL)	CBB6	203,182
RES	6,(IX+d)	DDCBddB6	221,203,XX,182
RES	6,(IY+d)	FDCBddB6	253,203,XX,182
RES	6,A	CBB7	203,183
RES	6,B	CBB0	203,176
RES	6,C	CBB1	203,177
RES	6,D	CBB2	203,178
RES	6,E	CBB3	203,179
RES	6,H	CBB4	203,180
RES	6,L	CBB5	203,181
RES	7,(HL)	CBBE	203,190
RES	7,(IX+d)	DDCBddBE	221,203,XX,190
RES	7,(IY+d)	FDCBddBE	253,203,XX,190
RES	7,A	CBBF	203,191

RES	7,B	CBB8	203,184
RES	7,C	CBB9	203,185,
RES	7,D	CBBA	203,185
RES	7,E	CBBB	203,187
RES	7,H	CBBC	203,188
RES	7,L	CBBD	203,189
RET		C9	201
RET	C	D8	216
RET	M	F8	248
RET	NC	D0	208
RET	NZ	C0	192
RET	P	F0	240
RET	PE	E8	232
RET	PO	E0	224
RET	Z	C8	200
RETI		ED4D	237,77
RETN		ED45	237,69
RL	(HL)	CB16	203,22
RL	(IX+d)	DDCBdd16	221,203,XX,22
RL	(IY+d)	FDCBdd16	253,203,XX,22
RL	A	CB17	203,23
RL	B	CB10	203,16
RL	C	CB11	203,17
RL	D	CB12	203,18
RL	E	CB13	203,19
RL	H	CB14	203,20
RL	L	CB15	203,21
RLA		17	23
RLC	(HL)	CB06	203,6
RLC	(IX+d)	DDCBdd06	221,203,XX,6
RLC	(IY+d)	FDCBdd06	253,203,XX,6
RLC	A	CB07	203,7
RLC	B	CB00	203,0
RLC	C	CB01	203,1
RLC	D	CB02	203,2
RLC	E	CB03	203,3
RLC	H	CB04	203,4
RLC	L	CB05	203,5
RLCA		07	7
RLD		ED6F	237,111
RR	(HL)	CB1E	203,30
RR	(IX+d)	DDCBdd1E	221,203,XX,30
RR	(IY+d)	FDCBdd1E	253,203,XX,30
RR	A	CB1F	203.31
RR	B	CB18	203,24
RR	C	CB19	203,25
RR	D	CB1A	203,26
RR	E	CB1B	203,27
RR	H	CB1C	203,28
RR	L	CB1D	203,29
RRA		1F	31
RRC	(HL)	CB0E	203,14
RRC	(IX+d)	DDCBdd0E	221,203,XX,14
RRC	(IY+d)	FDCBdd0E	253,203,XX,14
RRC	A	CB0F	203,15
RRC	B	CB08	203,8
RRC	C	CB09	203,9
RRC	D	CB0A	203,10
RRC	E	CB0B	203,11
RRC	H	CB0C	203,12
RRC	L	CB0D	203,13
RRCA		0F	15
RRD		ED67	237,103
RST	0	C7	199
RST	10h	D7	215
RST	18h	DF	223
RST	20h	E7	231
RST	28h	EF	239
RST	30h	F7	247
RST	38h	FF	255
RST	8	CF	207
SBC	A,(HL)	9E	158
SBC	A,(IX+d)	DD9Edd	221,158,XX
SBC	A,(IY+d)	FD9Edd	253,158,XX
SBC	A,A	9F	159
SBC	A,B	98	152
SBC	A,C	99	153
SBC	A,D	9A	154
SBC	A,E	9B	155
SBC	A,H	9C	156
SBC	A,L	9D	157
SBC	A,xx	DExx	222,XX
SBC	HL,BC	ED42	237,66
SBC	HL,DE	ED52	237,82
SBC	HL,HL	ED62	237,98
SBC	HL,SP	ED72	237,114
SCF		37	55
SET	0,(HL)	CBC6	203,198
SET	0,(IX+d)	DDCBddC6	221,203,XX,198
SET	0,(IY+d)	FDCBddC6	253,203,XX,198

SET	0,A	CBC7	203,199
SET	0,B	CBC0	203,192
SET	0,C	CBC1	203,193
SET	0,D	CBC2	203,194
SET	0,E	CBC3	203,195
SET	0,H	CBC4	203,196
SET	0,L	CBC5	203,197
SET	1,(HL)	CBCE	203,206
SET	1,(IX+d)	DDCBddCE	221,203,XX,206
SET	1,(IY+d)	FDCBddCE	253,203,XX,206
SET	1,A	CBCF	203,207
SET	1,B	CBC8	203,200
SET	1,C	CBC9	203,201
SET	1,D	CBCA	203,202
SET	1,E	CBCB	203,203
SET	1,H	CBCC	203,204
SET	1,L	CBCD	203,205
SET	2,(HL)	CBD6	203,214
SET	2,(IX+d)	DDCBddD6	221,203,XX,214
SET	2,(IY+d)	FDCBddD6	253,203,XX,214
SET	2,A	CBD7	203,215
SET	2,B	CBD10	203,208
SET	2,C	CBD1	203,209
SET	2,D	CBD2	203,210
SET	2,E	CBD3	203,210
SET	2,H	CBD4	203,212
SET	2,L	CBD5	203,213
SET	3,(HL)	CBDE	203,222
SET	3,(IX+d)	DDCBddDE	221,203,XX,222
SET	3,(IY+d)	FDCBddDE	253,203,XX,222
SET	3,A	CBDF	203,223
SET	3,B	CBD8	203,216
SET	3,C	CBD9	203,217
SET	3,D	CBDA	203,218
SET	3,E	CBDB	203,219
SET	3,H	CBDC	203,220
SET	3,L	CBDD	203,221
SET	4,(HL)	CBE6	203,230
SET	4,(IX+d)	DDCBddE6	221,203,XX,230
SET	4,(IY+d)	FDCBddE6	253,203,XX,230
SET	4,A	CBE7	203,231
SET	4,B	CBE0	203,224
SET	4,C	CBE1	203,225
SET	4,D	CBE2	203,226
SET	4,E	CBE3	203,227

SET	4,H	CBE4	203,228
SET	4,L	CBE5	203,229
SET	5,(HL)	CBEE	203,238
SET	5,(IX+d)	DDCBddEE	221,103,XX,238
SET	5,(IY+d)	FDCBddEE	253,203,XX,238
SET	5,A	CBEF	203,239
SET	5,B	CBE8	203,232
SET	5,C	CBE9	203,233
SET	5,D	CBEA	203,234
SET	5,E	CBEB	203,235
SET	5,H	CBEC	203,236
SET	5,L	CBED	203,237
SET	6,(HL)	CBF6	203,246
SET	6,(IX+d)	DDCBddF6	221,203,XX,246
SET	6,(IY+d)	FDCBddF6	253,203,XX,246
SET	6,A	CBF7	203,247
SET	6,B	CBF0	203,240
SET	6,C	CBF1	203,241
SET	6,D	CBF2	203,242
SET	6,E	CBF3	203,243
SET	6,H	CBF4	203,244
SET	6,L	CBF5	203,245
SET	7,(HL)	CBFE	203,254
SET	7,(IX+d)	DDCBffFE	221,203,XX,254
SET	7,(IY+d)	FDCBffFE	253,203,XX,254
SET	7,A	CBFF	203,255
SET	7,B	CBF8	203,248
SET	7,C	CBF9	203,249
SET	7,D	CBFA	203,250
SET	7,E	CBFB	203,251
SET	7,H	CBFC	203,252
SET	7,L	CBFD	203,253
SLA	(HL)	CB26	203,38
SLA	(IX+d)	DDCBdd26	221,203,XX,38
SLA	(IY+d)	FDCBdd26	253,203,XX,38
SLA	A	CB27	203,39
SLA	B	CB20	203,32
SLA	C	CB21	203,33
SLA	D	CB22	203,34
SLA	E	CB23	203,35
SLA	H	CB24	203,36
SLA	L	CB25	203,37
SRA	(HL)	CB2E	203,46
SRA	(IX+d)	DDCBdd2E	221,203,XX,46
SRA	(IY+d)	FDCBdd2E	253,203,XX,46

SRA	A	CB2F	203,47
SRA	B	CB28	203,40
SRA	C	CB29	203,41
SRA	D	CB2A	203,42
SRA	E	CB2B	203,43
SRA	H	CB2C	203,44
SRA	L	CB2D	203,45
SRL	(HL)	CB3E	203,62
SRL	(IX+d)	DDCBdd3E	221,203,XX,62
SRL	(IY+d)	FDCBdd3E	253,203,XX,62
SRL	A	CB3F	203,63
SRL	B	CB38	203,56
SRL	C	CB39	203,57
SRL	D	CB3A	203,58
SRL	E	CB3B	203,59
SRL	H	CB3C	203,60
SRL	L	CB3D	203,61
SUB	(HL)	96	150
SUB	(IX+d)	DD96dd	221,150,XX
SUB	(IY+d)	FD96dd	253,150,XX
SUB	A	97	151
SUB	B	90	144
SUB	C	91	145
SUB	D	92	156
SUB	E	93	147
SUB	H	94	148
SUB	L	95	149
SUB	xx	D6xx	214,XX
XOR	(HL)	AE	174
XOR	(IX+d)	DDAEdd	221,174,XX
XOR	(IY+d)	FDAEdd	253,174,XX
XOR	A	AF	175
XOR	B	A8	168
XOR	C	A9	169
XOR	D	AA	170
XOR	E	AB	171
XOR	H	AC	172
XOR	L	AD	173
XOR	xx	EExx	238,XX

APÊNDICE C

Códigos de Operação Z80

INSTRUÇÕES Op-code	FLAGS S	Z	—	H	—	P	N	C
ADC A,r	@	@	—	@	—	@	0	@
ADC HL,s	@	@	—	@	—	@	0	@
ADD A,r	@	@	—	@	—	@	0	@
ADD HL,s	—	—	—	@	—	—	0	@
ADD IX,s	—	—	—	@	—	—	0	@
ADD IY,s	—	—	—	@	—	—	0	@
AND r	@	@	—	1	—	@	0	0
BIT b,r	?	@	—	1	—	@	0	0
CALL pq	—	—	—	—	—	—	—	—
CALL c,pq	—	—	—	—	—	—	—	—
CCF	—	—	—	x	—	—	0	@
(A flag H assume o valor anterior da flag C)								
CP r	@	@	—	@	—	@	1	@
CPI	@	x	—	@	—	x	1	—
CPD	@	x	—	@	—	x	1	—
CPIR	@	x	—	@	—	x	1	—
CPDR	@	x	—	@	—	x	1	—
(Z passa a 1 se BC passa a 0,PV passa a 1 se A=(HL—1))								
CPL	—	—	—	1	—	—	1	—
DAA	@	@	—	@	—	@	—	@
DEC r	@	@	—	@	—	@	1	—
DEC s	—	—	—	—	—	—	—	—
DI	—	—	—	—	—	—	—	—
DJNZ e	—	—	—	—	—	—	—	—
EI	—	—	—	—	—	—	—	—
EX AF, AF'	—	—	—	—	—	—	—	—
EX DE,HL	—	—	—	—	—	—	—	—
EX (SP),HL	—	—	—	—	—	—	—	—
EX (SP),IX	—	—	—	—	—	—	—	—
EX (SP),IY	—	—	—	—	—	—	—	—
EXX	—	—	—	—	—	—	—	—

HALT	—	—	—	—	—	—	—	—
IM 0	—	—	—	—	—	—	—	—
IM 1	—	—	—	—	—	—	—	—
IM 2	—	—	—	—	—	—	—	—
INC r	@	@	—	@	—	@	0	—
INC s	—	—	—	—	—	—	—	—
IN A,(n)	—	—	—	—	—	—	—	—
IN r,(C)	@	@	—	@	—	@	0	—
INI	?	x	—	?	—	?	1	—
IND	?	x	—	?	—	?	1	—
(Z passa a 1 se B passa a zero)								
INIR	?	1	—	?	—	?	1	—
INDR	?	1	—	?	—	?	1	—
JP pq	—	—	—	—	—	—	—	—
JP c,pq	—	—	—	—	—	—	—	—
JP (HL)	—	—	—	—	—	—	—	—
JP (IX)	—	—	—	—	—	—	—	—
JP (IY)	—	—	—	—	—	—	—	—
JR e	—	—	—	—	—	—	—	—
JR c,e	—	—	—	—	—	—	—	—
LD(BC),A	—	—	—	—	—	—	—	—
LD A,(BC)	—	—	—	—	—	—	—	—
LD (DE),A	—	—	—	—	—	—	—	—
LD A,(DE)	—	—	—	—	—	—	—	—
LD I,A	—	—	—	—	—	—	—	—
LD R,A	—	—	—	—	—	—	—	—
LD A,I	@	@	—	0	—	x	0	—
LD A,R	@	@	—	0	x	0	—	—
LD SP,HL	—	—	—	—	—	—	—	—
LD SP,IX	—	—	—	—	—	—	—	—
LD SP,IY	—	—	—	—	—	—	—	—
LD r,r	—	—	—	—	—	—	—	—
LD s m n	—	—	—	—	—	—	—	—
LD A,(pq)	—	—	—	—	—	—	—	—
LD s,(pq)	—	—	—	—	—	—	—	—
LD (pq),A	—	—	—	—	—	—	—	—
LD (pq),s	—	—	—	—	—	—	—	—
LDI	—	—	—	0	—	x	0	—
LDD	—	—	—	0	—	x	0	—
(P/V passa a 0 se BC passa a 0)								
LDIR	—	—	—	0	—	0	0	—
LDDR	—	—	—	0	—	0	0	—
NEG	@	@	—	@	—	@	1	@
NOP	—	—	—	—	—	—	—	—
OR r	@	@	—	0	—	@	0	0
OUT (n),A	—	—	—	—	—	—	—	—
OUT (C),r	—	—	—	—	—	—	—	—
OUTI	?	x	—	?	—	?	1	—
OUTD	?	x	—	?	—	?	1	—
(Z passa a 1 se B passa a 0)								
OTIR	?	1	—	?	—	?	1	—
OTDR	?	1	—	?	—	?	1	—

POP AF	x	x	x	x	x	x	x	x
(Flags determinadas pelo byte no topo do stack)								
POP s	—	—	—	—	—	—	—	—
PUSH AF	—	—	—	—	—	—	—	—
PUSH s	—	—	—	—	—	—	—	—
RES b,r	—	—	—	—	—	—	—	—
RET	—	—	—	—	—	—	—	—
RET c	—	—	—	—	—	—	—	—
RETN	—	—	—	—	—	—	—	—
RETI	—	—	—	—	—	—	—	—
RLA	—	—	—	0	—	—	0	@
RL r	@	@	—	0	—	@	0	@
RLCA	—	—	—	—	—	—	—	—
RES b,r	—	—	—	—	—	—	—	—
RET	—	—	—	—	—	—	—	—
RET c	—	—	—	—	—	—	—	—
RETN	—	—	—	—	—	—	—	—
RETI	—	—	—	—	—	—	—	—
RLCA	—	—	—	0	—	—	0	@
RRCA	—	—	—	0	—	—	0	@
RLA	—	—	—	0	—	—	0	@
RRA	—	—	—	0	—	—	0	@
RLC r	@	@	—	0	—	@	0	@
RRC r	@	@	—	0	—	@	0	@
RL r	@	@	—	0	—	@	0	@
RR r	@	@	—	0	—	@	0	@
RRD	@	@	—	0	—	@	0	—
RLD	@	@	—	0	—	@	0	—
RST 00	—	—	—	—	—	—	—	—
RST 08	—	—	—	—	—	—	—	—
RST 10	—	—	—	—	—	—	—	—
RST 18	—	—	—	—	—	—	—	—
RST 20	—	—	—	—	—	—	—	—
RST 28	—	—	—	—	—	—	—	—
RST 30	—	—	—	—	—	—	—	—
RST 38	—	—	—	—	—	—	—	—
SBC A,r	@	@	—	@	—	@	1	@
SBC HL,s	@	@	—	@	—	@	1	@
SCF	—	—	—	0	—	—	0	1
SET b,r	—	—	—	—	—	—	—	—
SLA r	@	@	—	0	—	@	0	@
SRA r	@	@	—	0	—	@	0	@
SLR r	@	@	—	0	—	@	0	@
SUB r	@	@	—	@	—	@	1	@
XOR r	@	@	—	0	—	@	0	0

ÍNDICE

INTRODUÇÃO ... 9
PARA QUE SERVE O CÓDIGO-MÁQUINA? ... 11
PALAVRAS SOB A FORMA DE NÚMEROS ... 19
OS ENDEREÇOS E COMO LÁ CHEGAR... ... 30
TRABALHAR COM OS REGISTOS ... 37
- Instruções simples sobre registos ... 43

COMO FAZER SOMAS NOS REGISTOS ... 48
- Carga do conteúdo de uma posição de memória num registo ... 58
- Subtracção ... 61
- Operações sobre a flag «carry» ... 63

ASSEMBLERS, DISASSEMBLERS E PROGRAMAS DE DETERMINAÇÃO DE ERROS ... 65
- Assemblers ... 66
- Disassemblers ... 68
- Determinação de erros em programas ... 68
- Uso de um assembler ... 72
- Como introduzir o seu programa ... 74
- Disassemblers ... 81
  - Interrupção de código-máquina ... 83
  - Monitores ... 84
- Programas de determinação de erros (Debugging) ... 84
- Que programa? ... 87
- E finalmente... ... 87

COMO ESCREVER UM PROGRAMA ... 96
- A ideia geral ... 99
  - Fluxograma – geral ... 100
- Desenvolvimento da estrutura ... 100
- Fluxograma – Versão 2 ... 102
- Programa Basic ... 102
- Como somamos dois números de 16 bits ... 103
- Transferência de valores Basic para uso em código-máquina ... 109
- Como fazemos executar o programa em código-máquina ... 110
- Juntando tudo ... 116
- Conversão para linguagem Assembly ... 117
- Execução a «seco» ... 118
- Introdução do programa no Assembler ... 119
- Versão final ... 120

INSTRUÇÕES DE SALTO ... 123
- O registo PC ... 123

COMO USAR O TECLADO E O VISOR ... 136
O «STACK» ... 150
APÊNDICE A ... 161
APÊNDICE B ... 169
APÊNDICE C ... 185

Este livro acabou de se imprimir
em 1984
para a
EDITORIAL PRESENÇA, LDA.
na
Empresa Gráfica Feirense, Lda.
Vila da Feira