

65C02/6502B Assembler

Mark Andrews

Tradução
Rubens Gianesella

Revisão Técnica
Aluísio Mendes Vaz Filho
Engenheiro Civil - PUC
Pós-Graduado em Análise de Sistemas - PUC

McGraw-Hill
São Paulo
Rua Tabapuã, 1.105, Itaim-Bibi
CEP 04533
(011) 881-8604 e (011) 881-8528

Rio de Janeiro • Lisboa • Porto • Bogotá • Buenos Aires • Guatemala • Madrid • México • New York • Panamá • San Juan • Santiago

Auckland • Hamburg • Kuala Lumpur • London • Milan • Montreal • New Delhi • Paris • Singapore • Sydney • Tokyo • Toronto

Do original
Apple® Roots - Assembly Language Programming for Apple® IIe and Apple® IIc
Copyright © 1986 by McGraw-Hill, Inc.
Copyright © 1988 da Editora McGraw-Hill, Ltda.

Todos os diretos para a língua portuguesa reservados pela
Editora McGraw-Hill, Ltda.

Nenhuma parte desta publicação poderá ser reproduzida, guardada pelo sistema "retrieval" ou transmitida de qualquer modo ou por qualquer outro meio, seja este eletrônico, mecânico, de fotocópia, de gravação, ou outros sem prévia autorização, por escrito, da Editora.

Editor: Milton Mira de Assumpção Filho
Coordenadora de Revisão: Daisy Pereira Daniel
Supervisor de produção: José Rodrigues
Capa: layout: Cyro Giordano
arte-final: R. Iacobucci – Studio de Artes Gráficas

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

Andrews, Mark.
A581s 65C02/6502B Assembler / Mark Andrews ; tradução Rubens Gianesella ; revisão técnica Aluísio Mendes Vaz Filho. -- São Paulo : McGraw-Hill, 1988.

Bibliografia.

1. Apple IIe (Computador) – Programação
2. Apple IIe (Computador) – Programação 3. Assembler (Linguagem de programação para computadores) I. Título.

CDD-001.642
88-0112 -001.6424

Índices para catálogo sistemático:

1. Apple IIe : Computadores eletrônicos : Programação : Processamento de dados 001.642
2. Apple IIe : Computadores eletrônicos : Programação : Processamento de dados 001.642
3. Assembler : Linguagem de programação : Computadores : Processamento de dados 001.6424

SUMÁRIO

Sumário V
Introdução VII
Capítulo 1 Rompendo a barreira da linguagem Assembly 1
Capítulo 2 Sistemas numéricos 19
Capítulo 3 Por dentro do 6502B/65C02 32
Capítulo 4 Escrevendo e compilando um programa em linguagem Assembly 53
Capítulo 5 Rodando um programa em linguagem Assembly 85
Capítulo 6 Instruções do 6502B/65C02 104
Capítulo 7 Endereçando no Apple 135
Capítulo 8 "Loops" e Desvios 156
Capítulo 9 Manipulação bit a bit dos números binários 178
Capítulo 10 A matemática da linguagem Assembly 193
Capítulo 11 A mágica da memória 210
Capítulo 12 Princípios fundamentais dos gráficos Apple IIc/IIe 233
Capítulo 13 Controladores de jogos e o Apple Mouse 246
Capítulo 14 Gráficos Apple 268
Apêndice A Tabela de conversão de linguagem Assembly para linguagem de máquina 309
Apêndice B Tabela de conversão de linguagem de máquina para linguagem Assembly 316
Apêndice C Instruções do 65C02 321
Apêndice D Tabela de Op Codes do 65C02 323
Apêndice E Modos de endereçamento do 65C02 325
Apêndice F Instruções do 65802/65816 327
Apêndice G Modos de endereçamento do 65816 332
Apêndice H Tabela de Op Codes do 65816 335

Apêndice I Conjunto de caracteres ASCII para o Apple II 337
Bibliografia 343
Índice analítico 346



INTRODUÇÃO

Se o seu Apple não o compreende, talvez seja porque você não fale a linguagem dele; é preciso romper essa barreira de linguagem.

Este livro o ensinará a escrever programas em linguagem assembly – a mais rápida e de melhor aproveitamento da memória, dentre todas as linguagens de programação. Fornecerá conhecimentos sobre a linguagem de máquina, o idioma nativo de seu computador. Você poderá criar programas que, em BASIC ou outras linguagens menos avançadas, não seria possível.

Este livro, também, lhe mostrará que a programação em linguagem assembly não é tão difícil quanto imagina.

Muitos livros sobre linguagem assembly foram escritos, mas este é o primeiro a dedicar-se especificamente ao Apple IIc e ao Apple IIe. É, também, o primeiro a explicar como escrever programas em assembly utilizando o ProDOS, sistema operacional de disco do Apple IIc/IIe, que substituiu o antigo DOS 3.3. Aborda as características avançadas do microprocessador 65C02, o novo integrado do Apple IIc, que pode também ser instalado no Apple IIe.

Além disso, é o primeiro livro sobre linguagem assembly que mostra como utilizar os três assemblers mais populares para os Apple IIc e IIe: o pacote *"ProDOS Assembler Tools"* da Apple, o sistema assembler/editor *"Merlin Pro"* da Roger Warner Publishing e o assembler ORCA/M da The Byte Works, Albuquerque, New Mexico.

Os Apple IIc e IIe oferecem uma quantidade de novas características, de grande importância para os programadores. Entre elas, temos: apresentação de texto em 80

colunas, gráficos de dupla alta resolução e 64K de memória extra (todas elas estão na configuração BASICA do Apple IIc e são opcionais no IIe).

Tanto o IIc como o IIe apresentam teclados expandidos, incluindo novas teclas de função (maçã cheia e maçã vazia) e de caracteres especiais. Além disso, o Apple IIc possui um jogo especial de caracteres, projetados para utilização com mouse, sendo também disponível nos IIe equipados com cartões para mouse.

Tanto o IIc como o IIe estão sendo fornecidos com o ProDOS, sistema operacional de disco, que é o sucessor do DOS 3.3. O ProDOS não é simplesmente uma nova revisão do DOS 3.3; é um sistema operacional de disco totalmente novo, projetado para o IIc e IIe e para futuros computadores da linha Apple II. O ProDOS manuseia arquivos e disk drives de forma bem diferente da utilizada pelo DOS 3.3.

Uma comparação ponto por ponto entre o DOS 3.3 e o ProDOS está além do objetivo desta introdução. Devemos chamar a atenção, entretanto, para o fato de que *há tantas diferenças entre o ProDOS e o sistema que ele substitui, que a maioria dos programas em linguagem assembly, escritos sob o antigo sistema operacional de disco, simplesmente não rodará em um ambiente ProDOS*. Este é o primeiro livro sobre programação em assembly que inclui o ProDOS.

Tanto o Apple IIc como as mais recentes versões do IIe são, agora, equipados com um avançado microprocessador de 8 bits, o 65C02. Este novo membro da família 6502 é projetado para ser programado na linguagem 6502 padrão.

Mas o 65C02 contém diversas características novas. Além das 65 instruções usadas na linguagem assembly convencional do 6502, o 65C02 é equipado com várias instruções adicionais. Ele também apresenta modos de endereçamento não disponíveis nas primeiras séries de microprocessadores 6502.

Se você conhece BASIC - mesmo "BASICamente" – pode aprender a programar em assembly. Uma vez dominado o assembly, você poderá:

- Escrever programas que rodarão de 10 a 1000 vezes mais rapidamente que os programas em BASIC.
- Usar até 16 cores, simultaneamente, em qualquer dos modos gráficos do Apple IIc ou do Apple IIe – incluindo os gráficos de dupla alta resolução.
- Personalizar as apresentações na tela, misturando texto e gráficos da forma que desejar.
- Criar seus próprios jogos de caracteres.

Os progr
Apple ProDOS
leles, é interessant

O conhecimento embly também lhe permite criar música e efeitos sonoros para os pro ob ProDOS. Se c dquirir esse sistem programas que carregam e rodam automaticamente ao inserir o disco e lor, e misturar BASIC e assembly no mesmo programa, combinando a si Um pré-r ASIC com a velocidade e a versatilidade da linguagem assembly. ivro, é uma comp

Em outras palavra o a programar em assembly, você estará apto a começar a escrever pr o as mesmas técnicas dos programadores profissionais. Muitas dessas técnicas são impossíveis de se utilizar sem um conhecimento de linguagem assembly.

Finalmente, ao aprender a linguagem assembly, você estará aprendendo como os computadores funcionam; isto fará com que você programe melhor, em qualquer linguagem.

Ao ensinar a linguagem assembly, este livro irá fornecendo rotinas que poderão ser inseridas em programas escritos pelo usuário e uma biblioteca de programas tutoriais, interativos – especialmente projetados para eliminar o "grau de esforço" do aprendizado d bly.

lo 1 é uma introdução simples ao assembly. A característica principal
d a explicação clara dos números binários, também apresentando quatro
p IC que convertem números de uma base para outra.

ulo 3, você conhecerá os integrados 6502B/65C02, usados nos Apple
I ulo 4, você começará realmente a escrever programas em assembly. O
r resenta lições avançadas de programação e programas em assembly do
t ".

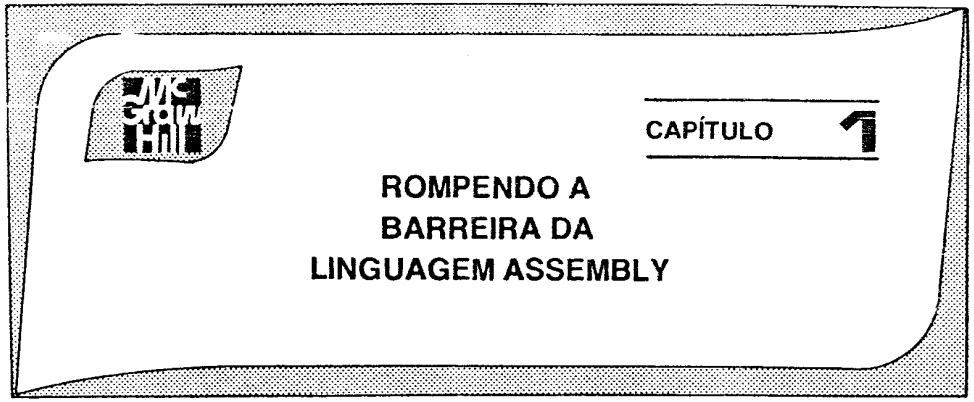
cê necessita em primeiro lugar, para utilizar este livro, é de um Apple
I televisor ou monitor (de preferência em cores) e pelo menos um disk
c sora é altamente recomendável, mas não essencial. Um controlador de
j u ambos) virão a calhar, bem como um segundo disk drive.

mas em assembly deste livro foram escritos utilizando três assemblers:
c Assembler, o Merlin Pro e o ORCA/M. Se você não possui nenhum
c e adquirir um antes de prosseguir. Todos os programas foram escritos
s seu Apple foi comprado antes da adoção do ProDOS, você precisará
a a e aprender a utilizá-lo.

equisito para o aproveitamento das lições de linguagem assembly, neste
l reensão básica do ProDOS, que pode ser conseguida através da leitura

do *ProDOS – Guia do Usuário*. Você deverá dominar, também, pelo menos os fundamentos do Applesoft BASIC ou de alguma linguagem de alto nível.

Há dois outros livros que você deve ler antes de começar a estudar a linguagem assembly: o manual do seu computador e um manual de consulta (*Reference Manual*) sobre seu computador. (A Apple publica manuais de consulta separados para o IIc e o IIe.) Outros livros úteis são: *Programming the 6502*, de Rodney Zaks, *Assembly lines: the book*, de Roger Wagner, e *6502 Assembly language programming*, de Lance A. Leventhal. Estes livros, além de outros, estão na Bibliografia.



CAPÍTULO 1

ROMPENDO A BARREIRA DA LINGUAGEM ASSEMBLY

Se você quer aprender a *linguagem assembly*, escolheu o livro certo. Com este volume e um Apple IIc ou IIe você pode começar já a programar em *linguagem de máquina*. Veremos a relação entre linguagem de máquina e linguagem assembly.

Ligue seu computador e digite o programa em BASIC, HI.TEST.BAS, listado como Programa 1.1. Rode esse programa e verá, imediatamente, porque ele se chama HI.TEST.BAS.

Programa 1.1
HI.TEST.BAS
Programa HI.TEST (Versão BASIC)

```
10  REM   *** HI.TEST.BAS ***
20  DATA  169,200,32,237,253,169,201,32,237,253,96
30  FOR L = 32768 TO 32778: READ X: POKE L,X: NEXT L
40  CALL 32768
```

LINGUAGEM DE MÁQUINA E LINGUAGEM ASSEMBLY

Como você pode ver, o Programa 1.1 está escrito em BASIC; entretanto, quando o digita e executa, seu computador rodará um programa em linguagem de máquina.

Note que é em linguagem de máquina, não em linguagem assembly. Como verá mais adiante, neste capítulo, a linguagem de máquina e a linguagem assembly são muito parecidas, mas não exatamente iguais. A linguagem de máquina é constituída de números –

nada além de números. Como a "manipulação de números" é o que melhor fazem os computadores, a linguagem de máquina é a ideal para eles.

De fato, a linguagem de máquina é, na realidade, a única que o computador compreende. Não importa em que linguagem um programa tenha sido escrito, ela deve ser convertida em linguagem de máquina para que o computador possa processá-la.

A razão principal da diferença entre a linguagem assembly e a de máquina é que a primeira foi projetada para seres humanos, não para máquinas. Do ponto de vista da estrutura, tanto quanto do vocabulário, a linguagem assembler é bem similar à de máquina.

De fato, a linguagem assembly não é uma linguagem de programação, mas simplesmente um sistema de notação projetado para facilitar a escrita de programas em linguagem de máquina.

Apesar de sua similaridade estrutural com a linguagem de máquina, um programa escrito em assembly apresenta-se bem diferente de um programa em linguagem de máquina.

Enquanto a linguagem de máquina consiste tão-somente de números, a linguagem assembly utiliza abreviaturas de três letras, denominadas *mnemônicos*. Desta forma, é mais fácil escrever programas em assembly do que em linguagem de máquina.

Em um aspecto, entretanto, a linguagem assembly é como qualquer linguagem de programação: antes que um programa em assembly possa ser rodado em um computador, deve ser convertido em linguagem de máquina. Por esta razão, programas escritos em assembly são, geralmente, chamados programas *código-fonte* e em linguagem de máquina, *código-objeto*.

Programas código-fonte em assembly são, usualmente, escritos com o auxílio de um tipo especial de pacote de software, chamado *editor assembler* ou simplesmente *assembler*.

Um pacote assembler, geralmente, inclui pelo menos dois tipos de programas utilitários: um *editor* de linguagem assembly, que permite ao usuário escrever programas em assembly, e um *assembler*, que converte programas escritos em assembly em linguagem de máquina.

Assembly e linguagem de máquina serão objeto de discussão mais detalhada ao longo deste capítulo.

COMO FUNCIONA O PROGRAMA HI.TEST.BAS

Estamos agora prontos para observar mais de perto o Programa 1.1. O programa HI.TEST.BAS começa com uma linha-título. A próxima linha, 20, é uma linha de dados, que equivalem a uma série de instruções em linguagem de máquina.

A linha 30 contém um loop que coloca os dados em linguagem de máquina (instruções) da linha 20 num bloco de RAM (termo definido logo adiante), que se estende do endereço de memória 32768 ao 32778, ou de $8000 a $800A, em notação hexadecimal. (Um endereço de memória – algumas vezes denominado *localização na memória* ou *registro de memória* – não é nada além de um número que pode ser usado para apontar a localização de qualquer elemento de dado ou *byte*, armazenado na memória de um computador.)

Existem 65.535 endereços de memória em um Apple IIe básico, e 131.070 num Apple IIc ou num IIe equipado com um cartão de 80 colunas expandido. Maiores informações sobre endereços de memória serão fornecidas adiante, especificamente no Capítulo 11, que enfoca a estrutura de memória do Apple IIc e IIe.

Finalmente, na linha 40, há uma instrução CALL, que executa o programa em linguagem de máquina recém-carregado na memória.

Para compreender o que o computador faz ao receber a instrução CALL da linha 40, será de grande valia um conhecimento básico da arquitetura dos microcomputadores, e de como seu Apple processa um programa em linguagem de máquina.

POR DENTRO DE UM MICROCOMPUTADOR

Um microcomputador pode ser dividido em três partes:

- Uma *unidade central de processamento* (CPU-Central Processing Unit). Como o próprio nome indica, é o componente central no sistema, onde todas as funções de computação se realizam. Todas as funções de uma unidade central de processamento estão contidas em uma *unidade microprocessadora* (MPU – Microprocessor Unit). A MPU do Apple – bem como a CPU – é um circuito com alta escala de integração (LSI – Large-scaled Integrated Circuit), um 6502B, se for um Apple IIe, ou um 65C02, se for um Apple IIC.

- Uma *memória*. Esta pode ser dividida em RAM (random access memory),

memória de acesso randômico, e ROM (read-only memory), memória apenas de leitura.

- Alguns *dispositivos de entrada e saída* (I/O, ou input/output devices). O dispositivo principal de entrada do computador é o *teclado*. O principal dispositivo de saída é o *monitor de vídeo*. Outros dispositivos aos quais o Apple pode ser conectado – ou, no jargão da informática, *interfaceado* – incluem impressoras, disk drives, modems, joysticks etc.

A Figura 1.1 é um diagrama de blocos que ilustra a arquitetura do Apple IIc e do Apple IIe. Neste capítulo, não nos ocuparemos com I/O; entretanto, teclado e tela serão abordados adiante, a partir do Capítulo 8.

A MEMÓRIA DO APPLE

A Figura 1.1 mostra os dois tipos de memória que um computador possui: RAM e ROM. A diferença importante entre elas é que a RAM pode ser modificada, mas não a ROM.

A ROM é permanentemente gravada num banco de integrados do Apple, estando sempre ali, com o computador ligado ou desligado; cada vez que o computador é desligado, porém, tudo o que se encontrava armazenado em RAM desaparece imediatamente.

O maior bloco de ROM no Apple estende-se do endereço de memória 53248 ($D000 em notação hexadecimal) ao 65535 ($FFFF em hexa). Uma quantidade de programas importantes encontra-se situada de forma permanente nesse bloco de ROM, incluindo o interpretador BASIC e o monitor de linguagem de máquina.

Programas em Linguagem de Máquina em RAM

Em livros de introdução aos computadores, um banco de RAM é, geralmente, comparado aos blocos de caixas postais existentes nas agências de correio.

Cada endereço num banco de RAM, da mesma forma que cada caixa postal, possui um número de identificação. Um programa de computador (como um funcionário ideal dos correios) pode acessar qualquer endereço num banco de RAM com a mesma facilidade.

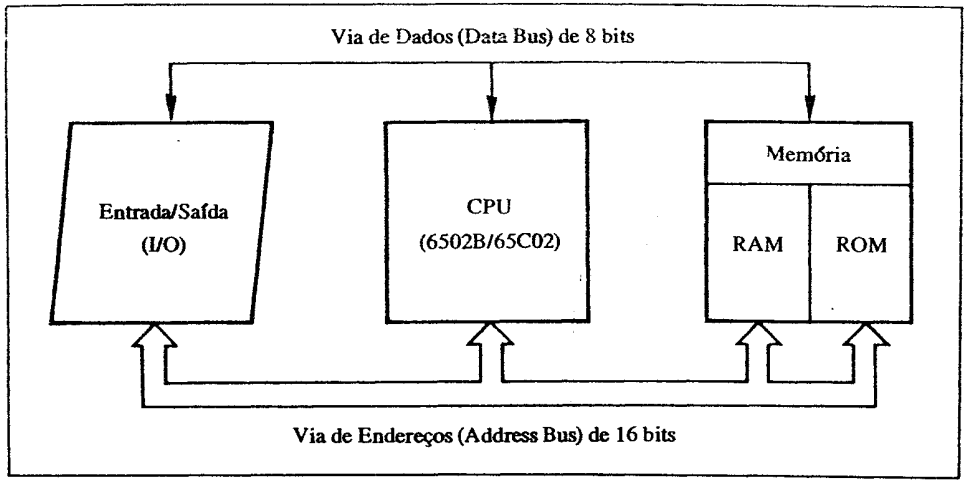


Figura 1.1. Diagrama de blocos de um microcomputador.

Em outras palavras, a informação armazenada em RAM pode ser recuperada de forma randômica, razão essa do nome RAM - random access memory.

O que acontece quando o computador processa um programa em linguagem de máquina? Cada programa em linguagem de máquina é formado por uma série de números; quando um desses programas é carregado na memória do computador, os números que o constituem são armazenados numa série de endereços em RAM.

O endereço inicial do bloco de memória no qual o programa está armazenado (conhecido como endereço de *origem* do programa) é, geralmente, armazenado numa localização especial da memória, predeterminada. Desta forma, ao chegar o momento de rodar o programa, seu endereço de início pode ser facilmente localizado.

Uma vez que um programa em linguagem de máquina tenha sido carregado em RAM e seu endereço de origem armazenado numa localização de memória acessível, há várias maneiras de iniciar a execução desse programa.

Por exemplo, um programa em linguagem de máquina armazenado em um Apple II pode ser executado utilizando-se uma instrução CALL, uma instrução USR(X), um comando travessão (–) do ProDOS ou um comando BRUN do ProDOS. Estes e outros métodos para rodar programas em linguagem de máquina serão explicados no Capítulo 5.

Processando Código Executável

Quando seu computador se dirige a uma localização de memória que foi identificada como sendo o endereço inicial de um programa, ele deve encontrar um bloco de código executável – isto é, o início de um programa em linguagem de máquina. Se ele encontrar o programa, executará a sua primeira instrução e, em seguida, irá para o endereço consecutivo na memória.

O computador repetirá este processo até que encontre o fim do programa ou uma instrução que lhe indique para pular para outro endereço.

A CPU DE SEU APPLE

Em um microcomputador, a CPU, geralmente, consiste de um único integrado (microprocessador). Os computadores Apple IIc e IIe utilizam tanto o 6502B quanto o 65C02.

O 6502B foi projetado para o Apple IIe e adotado, inicialmente, nesse equipamento. O 65C02 foi projetado para o Apple IIc, sendo utilizado até hoje. O 65C02 foi, também, adotado nos IIe, e é disponível como um opcional a ser instalado pelo usuário, para atualizar os antigos IIe.

Tanto o 6502B quanto o 65C02 são versões atualizadas e aperfeiçoadas do 6502, desenvolvido pela MOS Technology. O 6502 e outros integrados nele baseados são utilizados não somente pela Apple, mas em vários computadores pessoais produzidos pela Atari, Commodore e diversas outras empresas.

O 6502B utilizado no Apple IIe é, na realidade, apenas uma versão mais rápida do 6502 original. Já o 65C02 do Apple IIc e dos IIe mais recentes apresenta características extras em relação ao 6502. Além de ser mais rápido e consumir menos energia, ele reconhece uma série de instruções que o 6502 não compreende. O 65C02 apresenta também modos adicionais de endereçamento, que veremos num próximo capítulo.

Para nossos propósitos, entretanto, as similaridades entre o 6502B, o 65C02 e outros *chips* da família 6502, são mais importantes que as diferenças. Embora os diferentes números de série do 6502 possam causar alguma confusão, todos os membros da família 6502 foram projetados para ser programados utilizando-se o mesmo dialeto assembly, conhecido genericamente como linguagem assembly do 6502.

Uma vez que você tenha aprendido a escrever programas em linguagem assembly do 6502/65C02, estará apto a programar vários tipos de computadores pessoais, além de seu Apple, incluindo muitos dos produzidos pela Atari e Commodore.

Algo ainda mais importante: os *princípios* utilizados na programação em linguagem assembly da Apple são universais; são os mesmos princípios utilizados por todos os programadores em assembly, não importando o tipo de computador para o qual os programas sejam escritos. Aprendida a linguagem assembly do 6502/65C02, será fácil aprender a programar outros tipos de *chip* como o Z80 do Radio Shack e computadores baseados em CP/M, e mesmo os modernos e poderosos microprocessadores de 16 e 32 bits utilizados em computadores como o Macintosh da Apple e o IBM PC.

COMPILADORES, INTERPRETADORES E ASSEMBLERS

Após esta breve explicação sobre a arquitetura e o funcionamento básico do Apple, vamos observar mais de perto as relações entre as três categorias de linguagens de computador: linguagem de máquina, linguagem assembly e linguagens de alto nível.

Linguagens de alto nível não têm esse nome por serem particularmente complexas ou profundas, mas por estarem distanciadas da linguagem de máquina, a língua nativa do computador.

Há centenas, talvez milhares, de linguagens de alto nível, mas todas apresentam pelo menos uma característica em comum: possuem, no mínimo, uma ligeira semelhança com o Inglês. BASIC, por exemplo, compõe-se quase que exclusivamente de instruções – como PRINT, LIST, LOAD, SAVE, GOTO e RETURN – que são derivadas de palavras inglesas. A maioria das linguagens de alto nível também apresenta jogos de instruções baseada em palavras e frases da linguagem habitual.

Mas os computadores não entendem uma única palavra de Inglês; a única linguagem que compreendem é a linguagem de máquina, composta unicamente de números. Por esta razão, um programa escrito em qualquer outra linguagem tem de ser traduzido para linguagem de máquina, de forma que o computador possa compreendê-lo.

Como já foi mencionado, as pessoas que programam em assembly usualmente utilizam-se de produtos especiais de software, chamados assemblers, para converter a linguagem assembly em linguagem de máquina.

Da mesma forma, as pessoas que programam em linguagens de alto nível usam pacotes especiais de software, chamados *interpretadores e compiladores* para auxílio na

tradução dos programas para linguagem de máquina.

INTERPRETADORES E COMPILADORES

A diferença mais importante entre interpretadores e compiladores é que um interpretador destina-se a converter um programa em linguagem de máquina, durante a execução a cada vez que o programa é rodado, enquanto um compilador executa essa tarefa uma única vez.

Ao escrever um programa utilizando um interpretador, podemos salvar o programa em disco na sua forma original; o interpretador converte-lo-á em linguagem de máquina, sempre que ele for rodado. Utilizando-se, porém, um compilador, o programa deverá ser convertido em linguagem de máquina e, então, será salvo em disco como um programa em linguagem de máquina. Poderá, então, ser rodado como qualquer outro programa em linguagem de máquina, sem necessidade de se utilizar um compilador.

Os interpretadores são de utilização mais fácil, por serem projetados para se comportarem de forma "transparente ao usuário", isto é, nem mesmo se percebe que eles estão presentes.

O utilitário BASIC residente em seu Apple IIc ou IIe é um interpretador; utilizando-o, perceberemos quão transparente um interpretador BASIC pode ser. Quando escrevemos um programa em Applesoft BASIC, o interpretador residente no computador traduz cada linha de código, quando a escrevemos, para um tipo especial de linguagem, chamada linguagem *simbólica*. Cada vez que o programa é rodado, ele é, então, traduzido para linguagem de máquina.

Esta é uma forma cheia de meandros para rodar um programa, sendo um dos fatores que fazem do BASIC uma linguagem de execução lenta. Mas o processo é bem suave: ao rodar um programa em Applesoft BASIC, provavelmente nem mesmo percebemos o processo de tradução do BASIC para linguagem de máquina.

Uma vantagem dos interpretadores, com relação aos compiladores, é que os primeiros podem verificar cada linha do programa quanto a erros óbvios, assim que a linha é escrita. Se não verificam cada linha, apontam os erros assim que o programa é rodado, possibilitando a correção no local indicado.

Os compiladores são menos interativos: normalmente não verificam o programa quanto a erros, até que ele tenha sido compilado. Após o erro ter sido encontrado e corrigido, o programa tem de ser compilado novamente.

Os compiladores apresentam uma significativa vantagem sobre os interpretadores: produzem programas de execução mais rápida. Quando um programa é escrito utilizando-se um interpretador, ele tem de ser processado através do interpretador toda vez que for rodado. O compilador realiza o trabalho uma só vez, e não é mais necessária sua utilização para que o programa rode.

ASSEMBLERS E LINGUAGEM ASSEMBLY

A linguagem assembly, como já vimos, não é nem uma linguagem interpretada, nem uma linguagem compilada. A conversão de um programa em assembly para linguagem de máquina requer um editor/assembler (ou, simplesmente, assembler).

Dada a estreita relação entre assembly e linguagem de máquina, um assembler realiza uma tarefa bem mais fácil que a realizada pelos interpretadores e compiladores. Cada vez que estes convertem uma instrução em linguagem de máquina, um bloco – geralmente grande – de código de máquina tem de ser gerado.

As instruções usadas na linguagem assembly realizam funções mais simples que as usadas na maioria das linguagens de alto nível; desta forma, os programas código-fonte escritos em assembly tendem a ser bem mais longos que os escritos em linguagens de alto nível.

Entretanto, o jogo de instruções utilizado na linguagem assembly é extremamente versátil, uma vez que os mnemônicos podem ser combinados de forma praticamente ilimitada.

A linguagem assembly é, também, extremamente eficiente em termos de utilização de memória, já que os programas são criados por programadores humanos e não por máquinas geradoras de códigos.

Quando um programa é escrito em linguagem de alto nível, o resultado, é, usualmente, uma série de rotinas em linguagem de máquina, quase que desordenadamente colocadas umas após as outras. O interpretador ou o compilador, que traduzem tal programa em linguagem de máquina, repetem tipicamente a mesma seqüência de código, geralmente desperdiçando memória e tempo de processamento.

Em contraste, um bom programador em assembly usualmente escreverá um bloco importante de código uma só vez, durante o curso de um programa, e esse bloco será usado como uma sub-rotina, o que preserva a memória e reduz o tempo de processamento.

LINGUAGEM DE MÁQUINA: A LINGUAGEM DOS NÚMEROS

Para entender o que ocorre quando um programa em assembly é convertido em um programa em linguagem de máquina, é útil identificar algumas das diferenças e similaridades entre as duas linguagens.

A linguagem de máquina, em sua forma mais pura, é composta de *números binários* – números escritos como seqüências de "uns" e "zeros". O Programa 1.2 mostra como o programa HI.TEST, apresentado no início deste capítulo, mostrar-se-ia se escrito em linguagem de máquina pura, em números binários.

Programa 1.2
HI.TEST.BIN
O programa HI.TEST (Versão Código Binário)

```
10101001
11001000
00100000
11101101
11111101
10101001
11001001
00100000
11101101
11111101
01100000
```

Como podemos notar, não seria fácil escrever um programa em números binários. Felizmente, o faremos somente em raras ocasiões. Embora o computador veja cada programa em linguagem de máquina como uma série de números binários, ninguém realmente escreve um programa em notação binária, e sim num sistema de notação chamado *sistema hexadecimal.*

A despeito das aparências, os números hexadecimais são estreitamente relacionados com os números binários. Números em hexa são escritos não como seqüências de 1 e 0, mas como números arábicos convencionais, com as letras de A a F representando alguns valores extras.

Os números em hexa são escritos com combinações de algarismos e letras, porque, ao contrário dos números decimais, não têm como base o valor 10, e sim 16. As letras de A até F são usadas para representar os valores de 10 a 15.

Analisaremos melhor o sistema hexadecimal – e o porquê de sua utilização nos programas em assembly – a partir do Capítulo 2. Por enquanto, no Programa 1.3, temos o

aspecto do programa HI.TEST quando escrito em números hexadecimais.

Programa 1.3
HI.TEST.HEX
O programa HI.TEST (Versão Hexadecimal)

```
A9 C8
20 ED FD
A9 C9
20 ED FD
60
```

Os números em hexa no Programa 1.3 apresentam exatamente os mesmos valores dos números binários que vimos na versão binária do programa.

Você pode ainda não entender o significado das instruções do programa, mas pode perceber que a versão hexadecimal *parece* ser um pouco mais compreensível que a versão binária (e também mais fácil de ser digitada no computador).

LINGUAGEM ASSEMBLY: UMA LINGUAGEM DE MNEMÔNICOS

Já sabemos que a linguagem assembly é estreitamente relacionada com a linguagem de máquina; porém, a relação entre elas não é óbvia à primeira vista.

A linguagem assembly, diferentemente da linguagem de máquina, é escrita utilizando-se instruções de três letras, chamadas *mnemônicos*. Para o observador leigo, portanto, a linguagem assembly não se parece em nada com a linguagem de máquina.

Para cada instrução de três letras em assembly, existe uma instrução numérica cujo significado é exatamente o mesmo em linguagem de máquina.

Em outras palavras, geralmente existe uma relação direta entre os mnemônicos usados em um programa em assembly e as instruções numéricas da versão em linguagem de máquina desse mesmo programa.

Esta relação direta torna fácil a conversão de um programa em assembly para linguagem de máquina e vice-versa. Simplesmente trocamos cada instrução pela equivalente na outra linguagem. O Programa 1.4 demonstra a relação entre as linguagens.

Programa 1.4
HI.TEST.ASM

C programa HI.TEST (Versão Assembled)

```
COL. 1      COL. 2          COL. 3
Nº DE LINHA CÓDIGO-OBJETO   CÓDIGO-FONTE

1           A9 C8           LDA   #$C8
2           20 ED FD        JSR   $FDED
3           A9 C9           LDA   #$C9
4           20 ED FD        JSR   $FDED
5           60              RTS
```

CÓDIGO-OBJETO E CÓDIGO-FONTE

Se observarmos cuidadosamente as Colunas 2 e 3 do Programa 1.4, notaremos evidentes similaridades. Por razões que se tornarão claras mais adiante, as letras e números da Coluna 2 estão dispostos de forma ligeiramente diferente dos da Coluna 3, mas alguns padrões são os mesmos em ambas as colunas.

Na coluna 2, por exemplo, a instrução em linguagem de máquina A9 é usada duas vezes: a primeira na linha 1 e a segunda na linha 3. Na Coluna 3, o mnemônico em assembly LDA é também utilizado duas vezes – nas mesmas linhas e nas mesmas posições da instrução A9 em linguagem de máquina. Aparentemente, a instrução em código-objeto A9 equivale à instrução em código-fonte LDA. Conforme se pode deduzir, isto é verdade.

Consultando mais uma vez a listagem código-objeto da Coluna 2 veremos que a instrução 20 em linguagem de máquina também é usada duas vezes, em ambas sendo o equivalente, em código-máquina, da instrução JSR em código-fonte.

Até agora, efetuamos uma breve comparação entre assembly e linguagem de máquina; mais tarde, neste capítulo, discutiremos as similaridades e diferenças entre essas duas linguagens mais detalhadamente. Examinemos, então, o Programa 1.5, uma listagem em versão assembly do programa HI.TEST.

Programa 1.5
HI.TEST.5
O programa HI.TEST (Versão Código-Fonte)

```
LDA   #200
JSR   $FDED
LDA   #201
JSR   $FDED
RTS
```


O QUE FAZ O PROGRAMA HI.TEST.S

Como se vê, o HI.TEST.S é um programa bem pequeno, bem simples, em linguagem assembly. Contém apenas três mnemônicos – LDA, JSR e RTS – e três números – o número hexadecimal FDED e os números decimais 200 e 201.

O número 200 é um código de apresentação na tela (ou, simplesmente, código-tela) que equivale à letra H. O número 201 é o código-tela da próxima letra do alfabeto, a letra I. E o número hexadecimal FDED (65005 em notação hexadecimal) é o endereço do início de uma sub-rotina (residente no Apple) em linguagem de máquina, que provoca a apresentação de um caractere na tela.

No programa HI.TEST.S, os números 200 e 201 são precedidos do símbolo "#", e o número hexa FDED é precedido do símbolo "$". Na linguagem assembly do 6502/6502B/65C02, geralmente, quando o símbolo "#" precede um número, isto significa que esse número deve ser interpretado realmente como um número, não como um endereço de memória. Se, no programa HI.TEST.S, os números 200 e 201 não fossem precedidos do símbolo "#", eles seriam interpretados como endereços da memória do computador.

O outro símbolo especial no programa – o cifrão à frente do número FDED – é o prefixo, em linguagem assembly, para números hexadecimais. Se estamos familiarizados com a notação hexadecimal, reconheceremos como tal o número FDED. Porém, algumas vezes os números decimais e hexadecimais têm aparência idêntica. Este é o motivo da utilização do símbolo "$" para mostrar que um número deve ser tratado como hexadecimal.

Na linguagem assembly do Appie IIc/IIe, é possível usar o símbolo "#" e o "$" à frente de um mesmo número (contanto que o "#" esteja em primeiro lugar). Note, entretanto, que o símbolo "#" não é usado à frente do número $FDED no programa HI.TEST.S, uma vez que, nesse programa, $FDED deve ser interpretado como um endereço de memória, não como um número.

Como já foi mencionado, $FDED, no Apple, é o endereço de uma sub-rotina que apresenta um caractere na tela, sub-rotina essa chamada nas linhas 2 e 4 do HI.TEST.S.

Antes de compreender como o programa HI.TEST.S funciona, quando

convertido em linguagem de máquina, precisamos estudar um pouco o microprocessador do Apple, o 6502B ou o 65C02.

O integrado 6502B ou 65C02 é o coração – ou, mais precisamente, o cérebro – do computador. É um integrado bem complexo, mas possui somente sete componentes principais: uma *Unidade Lógica Aritmética* (ALU – Arithmetic Logical Unit) e seis *Registradores Internos*.

As funções e características desses componentes serão abordadas em outros capítulos. Para ajudar a compreender o funcionamento do programa HI.TEST.S, entretanto, segue uma rápida descrição de um registrador muito especial do 6502B/65C02, chamado acumulador.

O acumulador é um registrador mais atarefado do microprocessador. Antes que qualquer operação matemática ou lógica possa ser realizada em um número, em linguagem assembly do 6502B/65C02, o número tem de ser carregado no acumulador. A instrução em assembly geralmente utilizada para carregar um número no acumulador é LDA.

Vejamos a linha 1 do programa HI.TEST.S:

```
LDA #200
```

Nessa linha, o comando "LDA # 200" significa "Carregue o Acumulador (*Load* the *A*ccumulator) com o número 200". Em programação de computadores, um número pode ser usado para representar uma variedade de coisas. No programa HI.TEST.S, o número 200 representa a letra "H". Eis o porquê:

Um sistema especial, denominado ASCII, é geralmente utilizado para codificar letras, números e caracteres especiais em programas de computador. No sistema ASCII, cada letra, número e símbolo do teclado possui um número que o representa, em programas.

Através dos anos, o sistema ASCII tornou-se quase que um padrão na indústria de computadores. Entretanto, como os computadores Apple fazem uso de vídeo inverso, vídeo intermitente e outros efeitos especiais, a Apple utiliza um sistema ASCII modificado.

No sistema da Apple, o número 200 é um código ASCII para a letra "H", apresentada como maiúscula, em vídeo normal. O que a linha 1 do programa HI.TEST.S realmente significa, então, é "Carregue o acumulador com o código ASCII modificado da letra H maiúscula".


```
JSR $FDED
```

Na linguagem assembly do 6502B/65C02, o mnemônico JSR significa "Pule para Sub-rotina" (*J*ump to *Sub*routine). Esta instrução é utilizada de forma muito similar à instrução GOSUB do BASIC. Quando o mnemônico JSR é usado num programa em linguagem assembly, ele faz com que o programa pule para uma sub-rotina que começa no endereço de memória que segue a instrução JSR.

Em assembly, o mnemônico JSR é usualmente utilizado em conjunto com outro mnemônico, RTS, que significa "Volte da Sub-rotina" (*Re*turn from *S*ubroutine).

A instrução RTS também corresponde a uma instrução do BASIC, a RETURN. Quando uma instrução JSR é encontrada num programa em assembly, o endereço da próxima instrução do programa é, antes, colocado numa localização de fácil acesso num bloco especial de memória chamado *pilha* (stack). Então, o programa pula para o endereço da instrução JSR. Este endereço é, usualmente, o endereço de início de uma sub-rotina.

Quando uma sub-rotina é chamada com a instrução JSR, ela geralmente termina com uma instrução RTS. Quando essa instrução RTS é atingida, o endereço que tinha sido armazenado na pilha é recobrado. O programa retorna, então, a esse endereço e o processamento do corpo principal do programa continua.

A linha 2 do HI.TEST.S deve agora estar clara: o comando "JSR $FDED" significa "Pule para a sub-rotina que começa no endereço de memória $FDED". Esta sub-rotina apanha o código-tela que esteja no acumulador e automaticamente apresenta o caractere correspondente na tela, e a seguir devolve o comando ao programa em andamento.

Existe uma variedade de rotinas de I/O, similares a esta, presentes no Apple IIc/IIe. Usaremos algumas, ao longo deste livro.

```
LDA #201
```

Na linguagem assembly do 6502B/65C02, o número 201 é o código ASCII Apple para a maiúscula "I" normal. No programa HI.TEST.S, portanto, o comando "LDA#201" representa "Carregue o acumulador com o código ASCII Apple do I maiúsculo".

```
JSR $FDED
```

Este comando é idêntico ao da linha 2. Significa "Pule para a sub-rotina que começa no endereço de memória $FDED". Desta vez, entretanto, o acumulador foi

carregado com o valor 201, e a sub-rotina que inicia em $FDED causará a apresentação de um "I" na tela.

```
RTS
```

Quando uma instrução RTS é usada para finalizar uma sub-rotina, usualmente faz com que o programa pule para onde ele estava antes da chamada da sub-rotina. Neste caso, porém, a instrução RTS é usada para terminar o programa, não uma sub-rotina.

Quando RTS é usado para finalizar um programa completo, ele geralmente retorna o controle do computador a qualquer programa ou sistema que estava no controle, antes daquele programa iniciar. Se o programa HI.TEST.S do BASIC foi chamado, a instrução RTS da linha 5 transferirá o controle do computador ao interpretador BASIC.

DOIS PROGRAMAS ADICIONAIS

Até aqui, avançamos um bom passo neste capítulo introdutório: vimos a arquitetura dos microcomputadores, em geral, e a dos Apple IIc e IIe, em particular, comparamos a linguagem assembly com várias linguagens de alto nível e discutimos as formas pelas quais a linguagem assembly e as linguagens de alto nível são traduzidas para linguagem de máquina. Comparamos números decimais, hexadecimais e binários, e vimos como os números hexadecimais são usados nos programas em assembly. Mergulhamos no microprocessador, vimos como o código-fonte é transformado em código-objeto e observamos como o computador caminha através da memória ao processar um programa em linguagem de máquina. Realizamos até uma análise, linha por linha, de um pequeno programa em assembly, e vimos como um programa em linguagem de máquina pode ser chamado do BASIC sem ter de ser processado através de um assembler.

Vejamos, agora, outro programa em BASIC que faz uso de alguma linguagem assembly. O Programa 1.6 é chamado FLASH.BAS. Digite-o e rode-o, e você verá uma interessante apresentação na tela.

Programa 1.6
FLASH.BAS
Um programa em BASIC para "piscar" uma mensagem na tela

```
10  REM  ***  FLASH.BAS  ***
20  TEXT : HOME
30  PRINT : POKE 49164,1 : POKE 49166, 1: POKE 50,127
40  PRINT : PRINT "PISCANDO! O HOMEM APRENDEU CÓDIGO DE MÁQUINA!"
50  GOTO 50
```


COMO FUNCIONA O FLASH.BAS

Se você costuma programar em BASIC, provavelmente já escreveu vários programas que produzem mensagens piscantes na tela, como a apresentada pelo FLASH.BAS. No entanto, este pequeno programa em BASIC é pouco comum, pois não utiliza instruções em BASIC para gerar a intermitência na tela; ele o faz através de uma série bem estabelecida de comandos POKE.

No programa FLASH.BAS, os comandos POKE são usados para inserir números, diretamente, em três endereços de memória que o computador usa para gerar apresentações na tela.

Na linha 30, o comando POKE 49164,1 coloca o Apple no modo 40 colunas. Na mesma linha, o POKE 49166,1 assegura que o jogo principal de caracteres do computador esteja ativado e que o *jogo de caracteres alternativo* esteja desativado.

O último comando da linha 30 - POKE 50,127 - é o que aciona o modo *flashing* do computador. Agora o Apple está pronto para apresentar a mensagem piscante que aparece na linha 40. Finalmente, na linha 50, o programa "rola" num loop sem fim, o que evita que qualquer outra coisa seja apresentada na tela.

UM PROGRAMA PARA APRESENTAR AS FIGURAS DO MOUSE

Antes de passarmos para o Capítulo 2, eis mais um programa apresentado especialmente para os proprietários de Apple IIc e modelos recentes de Apple IIe. Não se trata de um programa em assembly ou linguagem de máquina, nem mesmo inclui instruções em linguagem de máquina. Mas provavelmente lhe interessará, caso seja proprietário de um Apple IIc, e uma versão desse programa em assembly será apresentada mais adiante.

Eis como o programa funciona: o Apple IIc/IIe tem dois jogos de caracteres – um normal e um alternativo; o alternativo varia conforme o modelo. Se o seu computador é um Apple IIc, ou um IIe com ROMs para mouse, o jogo de caracteres apresenta 32 figuras (*ícones*) do mouse.

No Apple IIc e no IIe atual, as figuras do mouse são o que se obtém quando o computador está em maiúsculas, no modo *flashing*. Mas se o seu Apple IIe não possui as ROMs *mousetext* instaladas, obterá o que normalmente se espera no modo *flashing* em maiúsculas: caracteres maiúsculos piscando.

Não é utilizado nenhum prefixo especial à frente de um número decimal; se um número sem prefixo aparece em um programa, ele é assumido como sendo um número decimal.

O exemplo a seguir ilustra a utilização dos prefixos para estabelecer a distinção entre números binários, hexa e decimais em programas em assembly:

%1101 = número binário 1101 (decimal 13)
$1101 = número hexadecimal 1101 (decimal 4353)
1101 = número decimal 1101

USANDO NÚMEROS BINÁRIOS

Os computadores só compreendem uma linguagem: a que é composta unicamente de números, chamada linguagem de máquina. A linguagem de máquina, em seu nível mais básico, consiste em números binários (1's e 0's). Logo, antes que os dados sejam introduzidos num computador, devem ser convertidos em seqüências de uns e zeros.

No sistema binário de notação, os 1 e 0 que compõem os números são conhecidos como *bits*. Uma série de quatro bits é chamada *nibble* (ou *nybble*); uma série de oito bits é chamada *byte;* e uma série de 16 bits é chamada *palavra* (embora existam também palavras de 8 bits).

Quando os bits e bytes que compõem um programa em linguagem de máquina são processados por um computador, eles são convertidos em séries de pulsos elétricos *on* e *off*. Dentro do computador, esses pulsos on/off correspondem, na realidade, à tensão nas várias linhas elétricas, alternando entre níveis altos e baixos.

Quando a tensão cai abaixo de um nível predeterminado, considera-se como uma chave aberta, estado esse representado como um 0 no sistema binário. Quando a tensão ultrapassa um determinado nível, considera-se como uma chave fechada, estado que representa um 1.

Vamos, agora, examinar uma série de números binários de 8 bits. Note que cada número binário que termina em 0 é o dobro do anterior, na lista:

00000001 = 1
00000010 = 2
00000100 = 4

00001000 = 8
00010000 = 16
00100000 = 32
01000000 = 64
10000000 = 128

Há ainda mais dois números que são notáveis, mas por razões completamente diferentes:

% 11111111 = 255
% 11111111 11111111 = 65535

O número %11111111, ou 255, é notável por ser o maior número de 8 bits, e o %11111111 11111111 é o maior número de 16 bits. (O espaço no centro do número %11111111 11111111 destina-se a facilitar a leitura. Geralmente são inseridos espaços também nos números de 8 bits, pela mesma razão. Algumas vezes veremos, por exemplo, o número binário 11111111 escrito na forma 1111 1111.)

O SISTEMA NUMÉRICO HEXADECIMAL

Como os computadores "pensam" em números binários, este sistema é excelente para representar dados de computação.

Os números binários, porém, apresentam um sério inconveniente: são de leitura extremamente difícil. Desta forma, o sistema binário não é o sistema numérico mais freqüentemente utilizado em programação assembly. O sistema numérico que encontraremos normalmente em programação assembly é o *hexadecimal.*

Da mesma forma que os números binários são baseados no valor 2, os números hexa são baseados no valor 16.

Os números hexa são comumente utilizados na programação em linguagem assembly por ajudarem a preencher o vazio existente entre o sistema binário e o decimal.

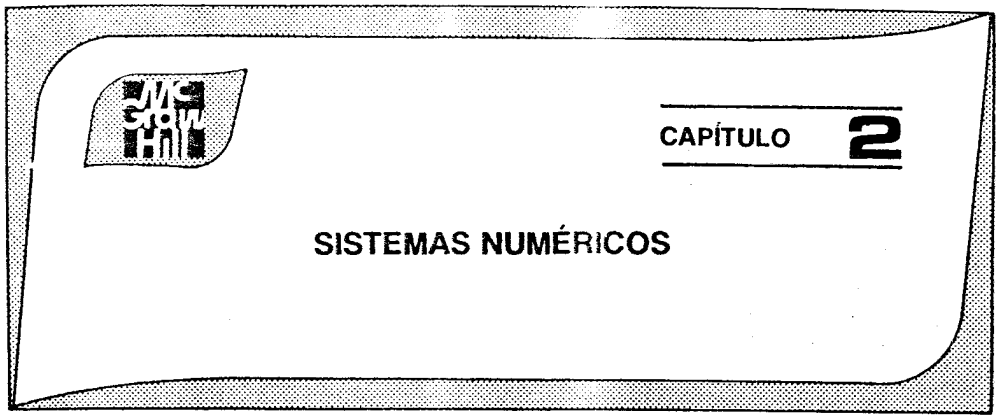
Como os números binários têm base 2 e os hexadecimais, base 16, uma série de quatro bits binários pode sempre ser traduzida em um dígito hexadecimal, uma série de 8 bits (um byte) pode ser representada por um par de dígitos hexadecimais e uma série de 16 bits (uma palavra), por um número hexadecimal de quatro dígitos.

Se seu computador é um Apple IIc ou um IIe recente, você poderá apresentar na tela todas as figuras do mouse, com este programa em BASIC.

Digite o MOUSETEXT.BAS, rode-o e divirta-se!

Programa 1.7
MOUSETEXT.BAS
Um programa em BASIC para apresentar as figuras do mouse

```
10  REM *** MOUSETEXT.BAS ***
20  PRINT CHR$ (4); "PR#3" : REM LIGAR FIRMWARE APERFEIÇOADO DE VIDEO
30  PRINT CHR$(27) ; CHR$(17) : REM ESC/CONTROL-Q (COLOCAR EM MODO 40 COLUNAS)
40  PRINT CHR$(15) : REM CONTROL-0 (COLOCAR EM MODO REVERSO)
50  PRINT CHR$(27) : REM TECLA ESC (LIGAR MOUSETEXT)
60  FOR L = 64 TO 95 : PRINT CHR$(L);:NEXTL : REM APRESENTAR AS FIGURAS DO MOUSE
70  PRINT CHR$(24) : REM CONTROL-X (DESLIGAR MOUSETEXT)
80  PRINT CHR$(14) : REM CONTROL-N (APRESENTAR CARACTERES NORMAIS)
```



CAPÍTULO 2

SISTEMAS NUMÉRICOS

Normalmente, as pessoas estão acostumadas a utilizar somente números decimais, que são baseados nos dígitos 0 a 9. Porém, algumas vezes, utilizamos o sistema de numeração romano, que usa letras para representar os números. Existem vários outros sistemas numéricos, diferentes do sistema decimal, como o chinês, o hebraico e o sânscrito.

Em programação de computadores, três sistemas numéricos são comumente utilizados:

- O sistema decimal, baseado no valor 10 e escrito utilizando-se os dígitos de 0 a 9.
- O sistema binário, baseado no valor 2 e escrito somente com dois dígitos de 0 e 1.
- O sistema hexadecimal, baseado no valor 16 e escrito utilizando-se os dígitos de 0 a 9, mais as letras de A a F.

PREFIXOS PARA AS BASES NUMÉRICAS

Quando um número binário aparece em um programa em linguagem assembly do Apple IIc ou IIe, o prefixo "%" é usado, pela maioria dos assemblers, para distingui-lo de um número decimal ou hexadecimal. Quando aparece um número hexadecimal, o prefixo "$" é o utilizado, pela maioria dos assemblers, para indicar que se trata de um número hexadecimal.

Não é utilizado nenhum prefixo especial à frente de um número decimal; se um número sem prefixo aparece em um programa, ele é assumido como sendo um número decimal.

O exemplo a seguir ilustra a utilização dos prefixos para estabelecer a distinção entre números binários, hexa e decimais em programas em assembly:

%1101 = número binário 1101 (decimal 13)
$1101 = número hexadecimal 1101 (decimal 4353)
1101 = número decimal 1101

USANDO NÚMEROS BINÁRIOS

Os computadores só compreendem uma linguagem: a que é composta unicamente de números, chamada linguagem de máquina. A linguagem de máquina, em seu nível mais básico, consiste em números binários (1's e 0's). Logo, antes que os dados sejam introduzidos num computador, devem ser convertidos em seqüências de uns e zeros.

No sistema binário de notação, os 1 e 0 que compõem os números são conhecidos como *bits*. Uma série de quatro bits é chamada *nibble* (ou *nybble*); uma série de oito bits é chamada *byte;* e uma série de 16 bits é chamada *palavra* (embora existam também palavras de 8 bits).

Quando os bits e bytes que compõem um programa em linguagem de máquina são processados por um computador, eles são convertidos em séries de pulsos elétricos *on* e *off*. Dentro do computador, esses pulsos on/off correspondem, na realidade, à tensão nas várias linhas elétricas, alternando entre níveis altos e baixos.

Quando a tensão cai abaixo de um nível predeterminado, considera-se como uma chave aberta, estado esse representado como um 0 no sistema binário. Quando a tensão ultrapassa um determinado nível, considera-se como uma chave fechada, estado que representa um 1.

Vamos, agora, examinar uma série de números binários de 8 bits. Note que cada número binário que termina em 0 é o dobro do anterior, na lista:

00000001 = 1
00000010 = 2
00000100 = 4

00001000 = 8
00010000 = 16
00100000 = 32
01000000 = 64
10000000 = 128

Há ainda mais dois números que são notáveis, mas por razões completamente diferentes:

% 11111111 = 255
% 11111111 11111111 = 65535

O número %11111111, ou 255, é notável por ser o maior número de 8 bits, e o %11111111 11111111 é o maior número de 16 bits. (O espaço no centro do número %11111111 11111111 destina-se a facilitar a leitura. Geralmente são inseridos espaços também nos números de 8 bits, pela mesma razão. Algumas vezes veremos, por exemplo, o número binário 11111111 escrito na forma 1111 1111.)

O SISTEMA NUMÉRICO HEXADECIMAL

Como os computadores "pensam" em números binários, este sistema é excelente para representar dados de computação.

Os números binários, porém, apresentam um sério inconveniente: são de leitura extremamente difícil. Desta forma, o sistema binário não é o sistema numérico mais freqüentemente utilizado em programação assembly. O sistema numérico que encontraremos normalmente em programação assembly é o *hexadecimal*.

Da mesma forma que os números binários são baseados no valor 2, os números hexa são baseados no valor 16.

Os números hexa são comumente utilizados na programação em linguagem assembly por ajudarem a preencher o vazio existente entre o sistema binário e o decimal.

Como os números binários têm base 2 e os hexadecimais, base 16, uma série de quatro bits binários pode sempre ser traduzida em um dígito hexadecimal, uma série de 8 bits (um byte) pode ser representada por um par de dígitos hexadecimais e uma série de 16 bits (uma palavra), por um número hexadecimal de quatro dígitos.

Na Tabela 2.1, os números decimais, hexadecimais e binários, de 1 a 16, são comparados. Examine atentamente a tabela e verá que estranhas combinações de letras e números, como "FC1C", "5DA4", ou até mesmo "ABCD", são números perfeitamente válidos no sistema hexadecimal.

Como podemos ver na tabela, o número decimal 16 é escrito como "10" em hexa e como "00010000" em binário, sendo assim um número redondo em ambos os sistemas. O dígito hexadecimal F, que vem logo após o hexa 10, é escrito como 00001111 em binário.

Ao nos tornarmos mais familiarizados com os sistemas binário e hexadecimal, começaremos a perceber muitas outras similaridades entre eles. Por exemplo: o número decimal 255 (o maior número de 8 bits) é 11111111 em binário e FF em hexa. O número decimal 65535 (o maior endereço de memória num computador de 64K) é escrito como FFFF em hexa e como 11111111 11111111 em binário.

Tabela 2.1 Comparação de números decimais, hexadecimais e binários.

Decimal	Hexadecimal	Binário
1	1	00000001
2	2	00000010
3	3	00000011
4	4	00000100
5	5	00000101
6	6	00000110
7	7	00000111
8	8	00001000
9	9	00001001
10	A	00001010
11	B	00001011
12	C	00001100
13	D	00001101
14	E	00001110
15	F	00001111
16	10	00010000

CONVERTENDO NÚMEROS DE UM SISTEMA EM OUTRO

Como são utilizados números decimais, hexadecimais e binários em programação assembly, seria prático usar-se algum tipo de ferramenta que convertesse números de um sistema em outro. Felizmente, existem várias ferramentas; eis algumas:

CONVERSORES BASEADOS EM SOFTWARE

O monitor de linguagem de máquina existente no Apple inclui um conversor decimal para hexadecimal e outro hexadecimal para decimal. Da mesma forma os possuem o assembler Merlin Pro e o utilitário depurador (debugging utility) Bugbyter, que acompanha o ProDOS assembler da Apple. Para maiores detalhes sobre estes utilitários, consulte o manual de consulta técnica (reference manual) do Apple IIc e IIe e os manuais que acompanham o Merlin Pro e o ProDOS assembler.

CALCULADORAS PARA PROGRAMADORES

A Texas Instruments produz uma calculadora, muito útil, chamada *Programmer*, que realiza conversão decimal/hexadecimal de forma praticamente instantânea, e também soma, subtrai, multiplica e divide tanto números decimais como hexadecimais. Muitos programadores em assembly utilizam, indispensavelmente, a Programmer da Texas ou outra calculadora similar.

TABELAS

Muitos dos livros sobre linguagem assembly contêm tabelas que podem ser consultadas para converter números de um sistema em outro. Há algumas dessas tabelas neste capítulo, além de uma série de programas em BASIC, que realizam automaticamente conversões decimal/hexadecimal, decimal/binário e binário/hexadecimal.

Vamos começar por um programa que converte números binários em decimais.

CONVERTENDO NÚMEROS BINÁRIOS EM DECIMAIS

Não é difícil converter um número binário em decimal. Em um número binário,

como já vimos, o bit mais à direita representa 2 à potência 0; o próximo bit, à esquerda deste, representa 2 à potência 1; o próximo, 2 à potência 2, e assim por diante.

Tabela 2.2. Valores dos Bits em um Número Binário de 8 Bits

Bit 0 = 2 à potência 0 =	1
Bit 1 = 2 à potência 1 =	2
Bit 2 = 2 à potência 2 =	4
Bit 3 = 2 à potência 3 =	8
Bit 4 = 2 à potência 4 =	16
Bit 5 = 2 à potência 5 =	32
Bit 6 = 2 à potência 6 =	64
Bit 7 = 2 à potência 7 =	128

Os dígitos, num número binário de 8 bits, são numerados de 0 a 7, da direita para a esquerda. O bit mais à direita – geralmente chamado de bit 0 – representa 2 à potência 0 ou o número 1. O bit mais à esquerda – geralmente chamado de bit 7 – vale 2 à potência 7 ou 128.

A Tabela 2.2 ilustra os valores de cada bit em um número binário de 8 bits.

A Tabela 2.3 apresenta um método simples de conversão de qualquer número binário de 8 bits para seu equivalente decimal. Em lugar de escrever o número da esquerda para a direita, vamos escrevê-lo em uma coluna vertical, com o bit 0 no topo da coluna e o bit 7 na base. Multiplicamos cada bit pelo valor que ele representa em decimal. Somamos os resultados dessas multiplicações, e o total será o valor decimal do número binário.

Tabela 2.3 Conversão de um Número Binário em Decimal

1 ×	1 = 1
0 ×	2 = 0
0 ×	4 = 0
1 ×	8 = 8
0 ×	16 = 0
1 ×	32 = 32
0 ×	64 = 0
0 ×	128 = 0
Total	= 41

Suponhamos, por exemplo, que desejamos converter o número binário 00101001 em decimal. A Tabela 2.3 mostra como a conversão pode ser efetuada.

CONVERTENDO NÚMEROS DECIMAIS EM BINÁRIOS

Vamos, agora, trabalhar em sentido inverso, e converter um número decimal em um número binário.

Primeiramente, vamos dividir o número por 2, e anotar o quociente e o resto. Como estamos dividindo por 2, o resto será 1 ou 0.

Em seguida, dividimos o quociente por 2, e escrevemos o resultado. Se há resto (1 ou 0), será escrito sob o resto da primeira operação.

Repetindo o procedimento até que não haja mais um número a dividir, escrevemos os restos obtidos, lendo-os de baixo para cima. Este será o número binário equivalente ao número decimal original.

Esta técnica de conversão é ilustrada na Tabela 2.4.

Para completar a conversão apresentada nessa tabela, simplesmente copiamos os dígitos binários da coluna da direita, escrevendo-os horizontalmente, da direita para a esquerda, com o dígito superior à direita. Veremos que o equivalente binário do número decimal (não hexadecimal) 117 é 01110101. Se você tiver uma tabela de conversão decimal → binário, poderá converter o resultado.

Tabela 2.4 Conversão de um Número Decimal em um Número Binário

117/2 = 58 com resto = 1
58/2 = 29 com resto = 0
29/2 = 14 com resto = 1
14/2 = 7 com resto = 0
7/2 = 3 com resto = 1
3/2 = 1 com resto = 1
1/2 = 0 com resto = 1
0/2 = 0 com resto = 0

CONVERTENDO NÚMEROS BINÁRIOS EM HEXADECIMAIS

Esta conversão de base numérica é ainda mais fácil. Para converter números hexadecimais para seus equivalentes binários, e vice-versa, simplesmente utilizamos a Tabela 2.5.

Para converter um número hexadecimal de múltiplos dígitos em um número binário, substituímos cada número ou letra, separadamente, pelo equivalente em binário, mostrado na Tabela 2.5. Por exemplo, o equivalente binário do número hexadecimal CO é 1100 0000. O equivalente binário do número hexa 7A1B é 0111 1010 0001 1011.

Para converter binários em hexadecimais, usamos a tabela de forma inversa: o número binário 1101 0110 1110 0101, por exemplo, é equivalente ao hexadecimal D6E5.

CONVERTENDO NÚMEROS DECIMAIS EM HEXADECIMAIS

É quase tão fácil converter números decimais em hexadecimais quanto decimais em binários.

Primeiramente, dividimos por 16 o inteiro decimal que desejamos converter. Escrevemos o resto, desta forma:

64540/16 = 4033 com resto 12

Dividimos a parte inteira do quociente por 16 e escrevemos o resultado:

4033/16 = 252 com resto 1

Repetimos este processo até obter quociente 0. Eis os cálculos necessários para converter o número decimal 64540 em um número hexadecimal:

64540/16 = 4033 com resto 12
4033/16 = 252 com resto 1
252/16 = 15 com resto 12
15/16 = 0 com resto 15

Após terminar esta série de cálculos, devemos converter todos os restos maiores de 9 no hexadecimal equivalente. Neste caso, três restos são maiores de 9: o valor 12 na primeira linha, o valor 12 na terceira linha e o valor 15 na quarta linha.

Tabela 2.5 Conversão Hexadecimal/Binário

Hexadecimal	Binário
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

O número decimal 12 equivale à letra C em notação hexadecimal, e o número decimal 15, à letra F. Assim, os restos do problema, convertidos em hexa, são:

C
1
C
F

Lendo os quatro números, partindo do inferior, teremos o número hexa FC1C, o equivalente que procurávamos, do número decimal 64540.

FAZENDO DO JEITO FÁCIL

Neste capítulo, comparamos três diferentes bases numéricas: o sistema decimal, o hexadecimal e o binário. Agora já sabemos como converter números em quaisquer dessas

bases. Algumas das técnicas de conversão são bem simples; outras, um pouco mais complicadas, exigindo uma memória fotográfica para sua utilização.

Felizmente, não é necessário ter tal memória fotográfica; simplesmente digitamos e salvamos o Programa 2.1, a seguir, e deixamos o computador trabalhar por nós.

O Programa 2.1, intitulado "MUDABASENUM", é um programa em BASIC, operado através de menus, que converte números entre quaisquer das bases estudadas neste capítulo. O programa assume que possuímos um display em 80 colunas.

No próximo capítulo, vamos dar uma olhada no interior do microprocessador de seu Apple, para ver como ele funciona. Então estaremos prontos para começar realmente a escrever programas em linguagem assembly.

Programa 2.1
MUDABASENUM
(Um Programa de Conversão Numérica em BASIC)

```
10  REM  ************************************************
20  REM ********* MUDABASENUM *********
30  REM ****** UM PROGRAMA DE CONVERSAO NUMERICA ******
40  REM     ************************************************
50  PRINT  CHR$ (4);"PR#3"
60  DIM HEX$(8),BIT$(8),H$(16),B$(16),TEMP$(2),BIT(8)
70  DATA  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
80  DATA  0000,0001,0010,0011,0100,0101,0110,0111
90  DATA   1000,1001,1010,1011,1100,1101,1110,1111
100  FOR L = 1 TO 16: READ H$(L): NEXT L
110  FOR L = 1 TO 16: READ B$(L): NEXT L
120  TEXT : HOME
130  PRINT : PRINT "MUDABASENUM:  UM PROGRAMA DE CONVERSAO DE BASE NUMERICA
140
150  PRINT : PRINT " ESTE PROGRAMA CONVERTE:"
160  PRINT : PRINT "(A) NUMEROS DECIMAIS EM HEXADECIMAIS"
170  PRINT "(B) NUMEROS HEXADECIMAIS EM DECIMAIS"
180  PRINT : PRINT "(C) NUMEROS BINARIOS EM DECIMAIS"
190  PRINT "(D) NUMEROS DECIMAIS EM BINARIOS"
200  PRINT : PRINT "(E) NUMEROS HEXADECIMAIS EM BINARIOS"
210  PRINT : "(F) NUMEROS BINARIOS EM HEXADECIMAIS"
220  PRINT : INPUT "QUE TIPO DE CONVERSAO VOCE DESEJA? (TECLE A-F): ";A$
230   IF A$ = "" THEN 220
240   IF  LEN (A$) <  > 1 THEN 220
250   IF A$ < "A" OR A$ > "F" THEN 220
260  A=ASC(A$) – 64 : REM TRADUZIR A$ EM UM INTEIRO DE 1(A) A 6(F)
270
280  REM ***** CONVERSAO DECIMAL EM HEXADECIMAL
290  TEXT:HOME:PRINT "CONVERSAO DECIMAL EM HEXADECIMAL (ALCANCE: 0 A 99999999)"
300  PRINT : PRINT "DIGITE UM INTEIRO DECIMAL POSITIVO(OU 'M' PARA O MENU):"
310   PRINT : INPUT "DEC: ";A$
320   FOR L = 1 TO 8:HEX$(L) = "": NEXT L
330   IF A$ = "M" THEN 120
340   FOR L = 1 TO 8:T$ =  RIGHT$ (A$,L)
350   IF  ASC (T$) < 48 OR  ASC (T$) > 57 THEN 310
360   NEXT L
370   IF  LEN (A$) < 1 OR  LEN (A$) > 8 THEN 310
380  N =  VAL (A$)
390  I = 8
400  TMP = N:N =  INT (N / 16)
410  TMP = TMP - N * 16
420   IF TMP < 10 THEN HEX$(I) =  RIGHT$ ( STR$ (TMP),1):
     GOTO 440
430  HEX$(I) =  CHR$ (TMP - 10 +  ASC ("A"))
440   IF N <  > 0 THEN I = I - 1: GOTO 400
450  PRINT "HEXA: ";
460   FOR L = 1 TO 8: PRINT HEX$(L);: NEXT L: PRINT
470   GOTO 310
480  REM ****** CONVERSAO HEXADECIMAL EM DECIMAL *****
490  TEXT : HOME : PRINT : PRINT "CONVERSAO HEXADECIMAL EM DECIMAL (ALCANCE: 0 A
     FFFFFFFF)"
500  PRINT : PRINT "DIGITE UM NUMERO HEXADECIMAL (OU 'M' PARA O MENU): ": PRINT
510  PRINT : INPUT "HEXA:"; A$
480  REM  ************* HEXADECIMAL-TO-DECIMAL CONVERSION
     **********************
490  TEXT : HOME : PRINT : PRINT "HEXADECIMAL-TO-DECIMAL
     CONVERSION (RANGE 0 TO FFFFFFFF)"
500  PRINT : PRINT "TYPE HEXADECIMAL NUMBER (OR 'M' FOR
     MENU):": PRINT
510  PRINT : INPUT "HEX: ";A$
520   IF A$ = "M" THEN 120
530   IF  LEN (A$) > 8 THEN 510
540  N = 0
550   FOR L = 1 TO  LEN (A$)
560  HEX$(L) =  MID$ (A$,L,1)
570   IF HEX$(L) < "0" OR HEX$(L) > "F" THEN 510
580   IF HEX$(L) <  = "9" THEN N = N * 16 +  VAL (HEX$(L)):
     GOTO 620
590   IF HEX$(L) < "A" THEN 510
600   IF HEX$(L) > "F" THEN 510
610  N = N * 16 +  ASC (HEX$(L)) -  ASC ("A") + 10
620   NEXT L
630   PRINT "DEC: ";N: PRINT
640   GOTO 510
650  REM ***** CONVERSAO BINARIO EM DECIMAL *****
660  TEXT : HOME : REM LIMPAR TELA
670  PRINT: PRINT "CONVERSAO BINARIO EM DECIMAL"
680  PRINT : PRINT "DIGITE UM NUMERO BINARIO DE 8 BITS (OU 'M' PARA O MENU):"
```

```
690  PRINT : INPUT "BIN: ";A$
700  IF A$ = "M" THEN 120
710  IF  LEN (A$) <> 8 THEN 690
720  FOR L = 8 TO 1 STEP  - 1
730 B$(L) =  MID$ (A$,L,1)
740  IF B$(L) <  > "0" AND B$(L) <  > "1" THEN 680
750  NEXT L
760  FOR L = 1 TO 8
770 BIT(L) =  VAL (B$(L))
780  NEXT L
790 ANS = 0
800 M = 256
810  FOR L = 1 TO 8
820 M = M / 2:ANS = ANS + BIT(L) * M
830  NEXT L
840  PRINT "DECIMAL:";ANS
850  GOTO 680
860  REM ***** CONVERSAO DECIMAL EM BINARIO"
870  TEXT: HOME : PRINT "CONVERSAO DECIMAL EM BINARIO (ALCANCE: 0-255)"
880  PRINT : PRINT "DIGITE UM INTEIRO POSITIVO (OU 'M' PARA O MENU):"
900  IF A$ = "M" THEN 120
910  IF  VAL (A$) < 0 OR  VAL (A$) > 255 THEN 890
920 NR =  VAL (A$)
930  FOR L = 8 TO 1 STEP  - 1
940 Q = NR / 2
950 R = Q -  INT (Q)
960  IF R = 0 THEN BT$(L) = "0": GOTO 980
970 BT$(L) = "1"
980 NR =  INT (Q)
990  NEXT L
1000  PRINT "BINARIO:";
1010  FOR L = 1 TO 8: PRINT BT$(L);: NEXT L: PRINT
1020  GOTO 880
1030  REM ***** CONVERSAO HEXADECIMAL EM BINARIO *****
1040  TEXT : HOME : PRINT "CONVERSAO HEXADECIMAL EM BINARIO (ALCANCE: 0 A FF)"
1050  PRINT : PRINT "DIGITE UM NUMERO HEXADECIMAL (OU 'M' PARA O MENU):"
1060  PRINT : INPUT "HEXA: "; A$
1070  IF A$ = "M" THEN 120
1080  IF  LEN (A$) > 2 OR  LEN (A$) < 1 THEN 1060
1090 HEX$(1) = "":HEX$(2) = ""
2000  FOR L = 1 TO  LEN (A$)
2010 HEX$(L) =  MID$ (A$,L,1)
2020  IF HEX$(L) < "0" OR HEX$(L) > "F" THEN 1060
2030  IF HEX$(L) > "9" AND HEXS$(L) < "A" THEN 1060
2040  NEXT L
2050  IF HEX$(2) = "" THEN HEX$(2) = HEX$(1):HEX$(1) = "0"
2060  FOR L = 1 TO 16: IF HEX$(1) = H$(L) THEN BIT$(1) =
    B$(L)
2070  NEXT L
2080  FOR L = 1 TO 16: IF HEX$(2) = H$(L) THEN BIT$(2) =
    B$(L)
2090  NEXT L
2095  PRINT "BINARIO: ";
2100  PRINT BIT$(1);BIT$(2): GOTO 1060
2110  REM ***** CONVERSAO BINARIO EM HEXADECIMAL *****
2120  TEXT : HOME : PRINT "CONVERSAO BINARIO EM HEXADECIMAL"
2130  PRINT : PRINT "DIGITE UM NUMERO BINARIO DE 8 BITS (OU 'M' PARAO MENU):"
2140  PRINT : INPUT "BIN: ";A$
2145  IF A$ = "M" THEN 120
2150  IF  LEN (A$) <  > 8 THEN 2140
2160  FOR L = 8 TO 1 STEP  - 1
2170 BIT$(L) =  MID$ (A$,L,1)
2180  IF BIT$(L) <  > "0" AND BIT$(L) <  > "1" THEN 2140
2190  NEXT L
2200 BIT$ = BIT$(1) + BIT$(2) + BIT$(3) + BIT$(4) + BIT$(5)
    + BIT$(6) + BIT$(7) + BIT$(8)
2210 T1$ =  LEFT$ (BIT$,4):T2$ =  RIGHT$ (BIT$,4)
2220  FOR L = 1 TO 16: IF T1$ = B$(L) THEN HEX$(1) = H$(L)
2240  NEXT L
2250  FOR L = 1 TO 16: IF T2$ = B$(L) THEN HEX$(2) = H$(L)
2260  NEXT L
2270  PRINT "HEXA: "; HEX$(1); HEX$(2)
2280  GOTO 2140
```

CAPÍTULO 3

POR DENTRO DO 6502B/65C02

Como vimos no Capítulo 1, o cérebro do computador é a unidade central de processamento ou CPU.

Uma unidade central de processamento, como o nome indica, é o componente central num sistema de computação – o componente no qual todas as funções de computação têm lugar. Em microcomputadores, como o Apple IIc e IIe, todas as funções da unidade central de processamento estão contidas num circuito de integração em larga escala (LSI), normalmente chamado de microprocessador ou unidade microprocessadora (MPU).

Originalmente, as MPUs utilizadas no Apple IIc e no Apple IIe eram ligeiramente diferentes. O Apple IIe foi originalmente projetado com base no microprocessador 6502B. Mas o Apple IIc e os novos IIe baseiam-se num integrado um pouco mais avançado, o 65C02. Ambos são membros da popular família 6502, de forma que os Apple IIc e IIe são quase 100% compatíveis.

O 65C02 pode rodar programas escritos para o 6502B, mas os programas escritos para o 65C02 não necessariamente rodarão nos 6502B. (Todos os programas deste livro rodarão em ambos os tipos.)

Além do 6502B e do 65C02, dois novos *chips* de 16 bits estão agora disponíveis para a série de computadores Apple II. Um deles, o 65802, pode processar dados a velocidades próprias da arquitetura de 16 bits, mas endereça somente 64K de espaço de memória – o mesmo espaço manuseado por um Aple II (64K) sem expansões.

O 65802 é totalmente compatível pino-a-pino com os 6502, 6502B e 65C02, de forma que pode ser encaixado diretamente nos Apple IIc e IIe. O 65802 roda o software

padrão do 6502 e também os novos softwares especialmente escritos para aproveitar as vantagens da capacidade de manuseio, em alta velocidade, de dados em 16 bits.

O outro novo *chip*, o 65816, tem uma via de dados (data bus) de 16 bits e uma via de endereços (address bus) de 24 bits, podendo manusear até 16 megabytes de memória. Ele gerencia a memória de forma mais complexa que a utilizada por outros *chips* da série 6502, e, desta forma, não é compatível pino-a-pino com o 6502B/65C02 do Apple IIe/IIc. Ele, entretanto, rodará o software projetado para a família 6502 de *chips* de 8 bits, além do software projetado especialmente para 65802 e 65816.

A arquitetura do 6502B e do 65C02 será discutida adiante neste capítulo, e os dois novos *superchips* serão abordados mais detalhadamente ao longo do livro.

Mas antes de podermos compreender a operação de qualquer microprocessador – mesmo o 6502 sem sofisticações, usado no primeiro Apple II – temos de conhecer algo a respeito da operação dos *chips* em geral; é necessário ter uma noção geral do que ocorre dentro dos *chips* quando processam dados e como acessam dados armazenados na memória do computador.

Comecemos por discutir como a CPU do Apple localiza dados armazenados na memória.

A MEMÓRIA DO APPLE

Já sabemos que RAM é a memória de curta duração no computador, enquanto a ROM é a memória de longa duração. No jargão da informática, diz-se que a RAM é *volátil*, e a ROM é *não-volátil*. Isto significa que a RAM pode ser alterada (ou perdida), o que não ocorre com a ROM.

RAM é o espaço livre da memória do computador. Quando ligamos o computador, a RAM é como uma folha de papel em branco. Podemos armazenar nela o que desejamos, incluindo texto, dados, programas ou até mesmo figuras a serem apresentadas na tela (tudo isto, é claro, representado por números).

Quando o computador é desligado, entretanto, tudo o que estava armazenado em RAM desaparece. Isto é o que mantém no mercado os fabricantes de discos para computadores*.

* Quando carregamos um programa em disco na memória, estamos colocando-o em RAM, a mesma que é perdida ao desligar-se o computador.

Já efetuamos uma comparação entre os bancos de RAM num microcomputador e as caixas postais nas agências dos correios. Dentro do computador, cada uma dessas "caixas postais" é chamada de registro de memória. E cada registro de memória, assim como cada caixa postal, tem um único endereço de memória.

Num Apple IIc ou IIe, cada endereço de memória pode conter somente um número de 8 bits – isto é, um número entre 0 e 255. Este número pode representar somente:

- um número propriamente dito;
- um código que representa um caractere de texto;
- uma instrução em linguagem de máquina;
- um elemento de dado (como uma parte de figura gráfica).

Uma vez que um computador contém muitos registros de memória, e o valor armazenado em cada registro pode ter vários significados, é necessário dizer ao computador duas coisas antes que ele possa rodar um programa: onde o programa está situado na memória e se o valor em cada número do programa deve ser interpretado como um número, um caractere de texto, uma instrução em linguagem de máquina ou um elemento de dado.

RODANDO UM PROGRAMA EM LINGUAGEM DE MÁQUINA

Para poder rodar um programa em linguagem de máquina, o computador precisa saber o *endereço de início* do programa – a localização na qual o programa foi armazenado na memória. Sabendo onde o programa começa, o computador pode ir diretamente à primeira instrução e executá-la. O computador irá, então, mover-se para o próximo endereço de memória, onde, se o programa foi adequadamente escrito, encontrará outra instrução.

Ao escrever programas em assembly com um dos três assemblers utilizados neste livro, normalmente indicaremos a localização de memória em que um programa começa, digitando uma linha como esta:

```
ORG $8000
```

Num programa em assembly, uma linha como esta é chamada *diretiva de origem*

(ORG – origin directive). O assembler Apple ProDOS requer a utilização de uma diretiva ORG, mas poderemos dispensá-la se possuirmos um assembler Merlin Pro ou ORCA/M, pois ambos assinalam um endereço default para o programa, se não assinalemos nenhum endereço.

Quando o assembler encontra uma linha ORG no início de um programa, ele montará o código-fonte que segue em um programa em linguagem de máquina que começa no endereço dado na diretiva; neste caso, no endereço hexadecimal $8000(ou 32768 em notação decimal). Uma vez que o programa tenha sido montado e o endereço dado na diretiva ORG – ou em um endereço predeterminado – ele poderá ser salvo em disco e mais tarde rodado pela utilização de um comando ProDOS tão simples quanto

```
BRUN PATHNAME
```

onde a palavra PATHNAME representa o nome completo do programa.

USANDO DADOS EM UM PROGRAMA EM LINGUAGEM ASSEMBLY

Quando uma linha ORG é utilizada em um programa em assembly, ela deve apontar ao primeiro endereço do programa que contém código executável (código de máquina gerado por instruções em linguagem assembly válidas).

Uma linha ORG nunca deve apontar para um segmento de programa composto por código de máquna não-executável, como, por exemplo, uma tabela de dados. Se isto acontecer, o computador tentará interpretar os dados como código executável e executá-lo; os resultados serão imprevisíveis.

Isto não significa que não poderemos usar tabelas de dados em programas em assembly; é claro que isto é possível. Apenas os blocos de dados não-executáveis devem ser usualmente armazenados em blocos separados de memória. Assim, esses dados podem ser acessados, conforme forem necessários, do corpo principal do programa, sem haver mistura com o código executável, durante a execução do programa.

NÚMEROS DE 8 BITS E DE 16 BITS

O Apple IIc e o Apple IIe pertencem, ambos, à categoria dos computadores ditos *de 8 bits*. Isto porque a série Apple II foi originalmente projetada com base no microprocessador 6502, um *chip* de 8 bits.

Os microprocessadores de 8 bits podem processar números binários de até 8 casas, somente. Como vimos no capítulo anterior, o 6502/6502B/65C02 pode processar cálculos com um número binário não maior que 255. De fato, ele não pode nem mesmo realizar um cálculo cujo resultado seja maior que 255!

Obviamente, esta limitação em 8 bits torna muito inconveniente a manipulação de grandes números. De fato, um 6502/6502B/65C02 é como uma calculadora que não pode trabalhar com um número maior que 255.

Para trabalhar com números acima de 255, um computador de 8 bits tem de realizar uma série bem complexa de operações. Se um número é maior que 255, um computador de 8 bits tem de desmembrá-lo em porções de 8 bits e realizar cada operação necessária em cada número de 8 bits. Após isto, o computador precisa agregar novamente esses números de 8 bits.

Se o resultado de um cálculo é maior que 8 bits, as coisas são ainda mais complicadas, pois os registros de memória no Apple IIc e IIe são também incapazes de manusear números maiores que 255. Cada célula na memória RAM do Apple, bem como na memória ROM, é um registro de memória de 8 bits. Assim, para armazenar um número maior que 255, o computador precisa desmembrá-lo em dois ou mais números de 8 bits e armazenar cada um desses números em uma localização separada de memória. Ao precisar consultar o número original, será necessário juntar os "pedaços" de 8 bits, novamente.

A ARQUITETURA DE MEMÓRIA DOS MICROPROCESSADORES 6502B/65C02

Chegamos, agora, num ponto que pode causar alguma confusão. Embora os Apple IIc e IIe não possam processar números de mais de 8 bits, podem manusear *endereços* de até 6 bits.

A Figura 3.1, um diagrama de blocos simplificado da CPU do computador e componentes a ela conectados, pode ajudar a compreensão deste ponto.

Ao examiná-la, veremos que o 6502B/65C02 tem:

- seis registros internos (PC, SP ou S, P, X, Y e A);
- uma unidade aritmética lógica (ALU);
- uma *via de dados* de 8 bits e uma *via de endereços* de 16 bits.

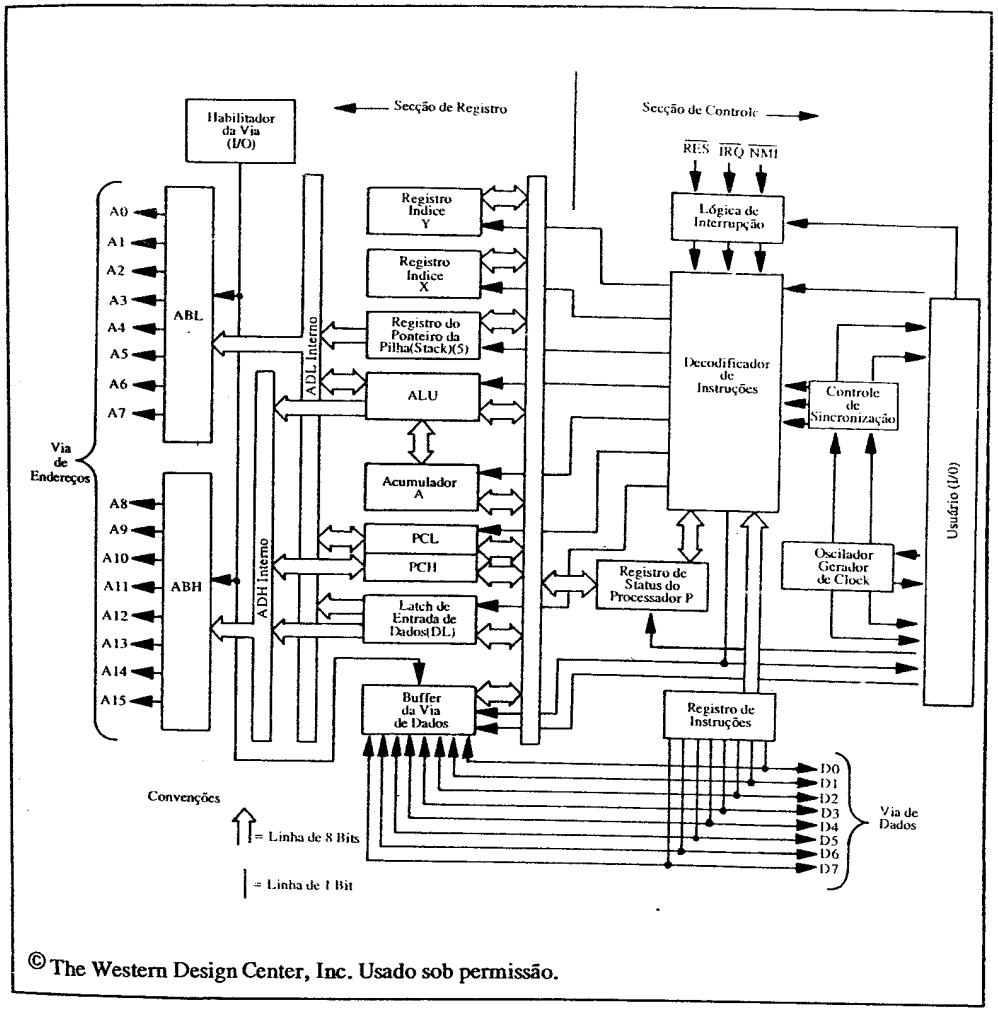


Figura 3.1 Diagrama de Blocos do Microprocessador 6502B/65C02.

A Figura 3.1 inclui, também, um bloco que representa os dispositivos de entrada e saída (I/O devices) e um bloco que representa a memória (RAM e ROM). Os dispositivos I/O serão abordados no Capítulo 11. Nesta secção, abordaremos as porções da Figura 3.1 críticas para a memória: as vias de dados e de endereços e o *contador de programa (program counter).*

VIAS DE DADOS E DE ENDEREÇOS

Na parte inferior da Figura 3.1 há uma série de linhas, com a indicação "via de dados". A margem esquerda da figura apresenta uma fileira de flechas com a indicação "via de endereços". Na terminologia de computação, uma via é uma linha pela qual a informação é transmitida dentro do computador. As barras na Figura 3.1 representam linhas de transmissão de dados e endereços dentro do Apple IIc ou IIe.

Há algo importante a respeito das duas vias ilustradas na Figura 3.1; a via de dados é uma via de 8 bits e a via de endereços é uma via de 16 bits, podendo transmitir, respectivamente, 8 e 16 bits por vez.

Desta forma, como o 6502B/65C02 tem uma via de dados de 8 bits, não pode manipular parcelas de dados com mais de 8 bits; porém, por ter uma via de endereços de 16 bits, pode controlar endereços de memória de até 16 bits. Vejamos porquê.

O CONTADOR DE PROGRAMA DO 6502B/65C02

Na região central da Figura 3.1 há um campo dividido em duas partes, com as siglas PCL (Program Counter-Low, ou Contador de Programa-Baixo e PCH (Program Counter-High, ou Contador de Programa-Alto). Este campo duplo representa um registro interno chamado Contador de Programa (PC). Na família de integrados 6502, o contador de programa é um registro usado para acompanhar os endereços dos registros de memória usados por um programa. Quando o computador roda um programa em linguagem de máquina, o contador de programa da CPU sempre retém o endereço do próximo byte de dados a ser processado. Cada vez que um byte é processado, o contador é, automaticamente, incrementado e, então, mantém o endereço do próximo byte a ser processado.

O campo na Figura 3.1 foi dividido em PCL e PCH, porque, na realidade, o contador de programa na família 6502 é formado por dois registros de 8 bits, sempre utilizados em conjunto, com um registro de 16 bits.

Este é o motivo pelo qual o computador pode manipular endereços de até 16 bits: ele simplesmente usa dois registros de 8 bits como contador de programa. Assim, ele pode endereçar na extensão de $0000 a $FFFF (ou, em notação decimal, de 0 a 65535). Como vimos no Capítulo 2, o equivalente binário de $FFFF, ou 65535, é 1111 1111 1111 1111 – o maior número de 16 bits.

Agora já sabemos o que as pessoas querem dizer, quando se referem a vias de

dados de 8 bits e vias de endereços de 16 bits. E sabemos também porque a maioria dos computadores de 8 bits existentes no mercado pode acessar 64K de memória.

CHAVEAMENTO DE BANCOS

Se o computador é um Apple II e com 64K de memória, isto é tudo o que precisamos saber, por ora, sobre a forma pela qual sua CPU acessa dados na memória.

Se for um Apple IIc ou um IIe com 128K de memória há mais um ponto a observar: como os projetistas de computadores conseguiram colocar 128K de memória em 64K de RAM.

Eles o fizeram através de uma técnica denominada *chaveamento de bancos.*

Chaveamento de bancos não é um conceito difícil; consiste em poder-se alternar, entre diferentes blocos de memória, num mesmo espaço de endereçamento. Se for um Apple IIc ou um Apple IIe expandido, ele possui dois blocos de 64K de memória, um deles chamado *memória principal* e outro *memória auxiliar*. Para alternar entre esses dois blocos de memória, é preciso mudar o conteúdo de determinada série de localizações de memória (especificamente, as localizações compreendidas entre os endereços de memória $C003 e $C016). Essas localizações são algumas vezes denominadas *soft switches,* por serem usadas exatamente como chaves (hard switches) para ligar e desligar algo.

Se não fosse utilizado o chaveamento de bancos, o Apple IIc (ou IIe expandido) teria capacidade de memória de somente 64K. Com o chaveamento, ele pode acolher 128K de dados, embora não seja um verdadeiro computador de 128K. Na realidade, apresenta-se como dois computadores de 64K acoplados, dispondo de um teclado em comum.

Como sabemos usar os *soft switches* adequados para efetuar o chaveamento entre a memória principal e a auxiliar, poderemos nos movimentar livremente entre elas. Porém, nunca poderemos acessar ambos os bancos de memória simultaneamente. (Como o chaveamento ocorre rapidamente, não é usualmente perceptível pelo usuário; o programador, entretanto, deve considerá-lo.)

As técnicas de chaveamento de bancos podem também ser utilizadas para chavear certos segmentos da memória principal de ROM para RAM, e de volta a ROM. Estas operações serão descritas mais detalhadamente no Capítulo 11, dedicado à organização da memória do Apple IIc e do Apple IIe. Por ora, é suficiente compreender

que as técnicas de chaveamento de bancos expandem a capacidade de memória do Apple IIc, de 64K para 128K e, ainda mais, no Apple IIe.

A CPU DE SEU APPLE

Observemos novamente a Figura 3.1. Os microprocessadores 6502B e 65C02 contêm, como já mencionado, sete partes principais: seis registros endereçáveis e uma unidade aritmética lógica ou ALU. (Os novos *chips* 65802 e 65816 serão discutidos, juntamente com o endereçamento, na conclusão deste capítulo e no Capítulo 7.)

A UNIDADE LÓGICA ARITMÉTICA

Da mesma forma que o centro de comando do computador é o microprocessador, o centro de comando da CPU é a sua ALU. Cada vez que um *chip* do tipo 6502 realiza um cálculo ou operação lógica, a ALU é o componente do *chip* onde o trabalho realmente ocorre.

A ALU pode realizar, na verdade, somente dois tipos de cálculo: adição e subtração. Divisão e multiplicação podem também ser realizadas pela ALU, mas somente como seqüências de operações de adição e subtração.

A ALU pode também comparar valores, uma vez mais através de subtrações. Para comparar dois valores, a ALU simplesmente subtrai um do outro, verificando se um deles é maior ou se ambos são iguais.

O ACUMULADOR

Na Figura 3.1, a ALU aparece logo acima do acumulador. Quando dois números devem ser adicionados, subtraídos ou comparados, um deles é primeiramente armazenado no acumulador do 6502B/65C02; depois, o acumulador deposita esse número na ALU, através de uma das entradas da ALU. O outro número é, então, colocado na ALU através da outra entrada. Finalmente, a ALU realiza a operação requisitada, e o resultado aparece na saída da ALU. Assim que a resposta aparece, é colocada no acumulador, onde substitui o valor originalmente ali armazenado.

O Programa 3.1, chamado ADDNRS, é um pequeno programa em linguagem assembly que demonstra como funciona esse processo.

Programa 3.1
O PROGRAMA ADDNRS
Um Exemplo de Operação na ALU

```
LDA #2
ADC #3
STA $0300
```

A primeira instrução do programa, LDA, significa "Carregue o acumulador" (LDA = Load the accumulator). Quando a instrução LDA é encontrada em um programa, o acumulador é carregado com um determinado número: especificamente, o valor do operando que segue a instrução. No programa ADDNRS, o efeito da instrução LDA é o de carregar o acumulador com o número 2. (O símbolo "#" à frente do numeral 2 indica que esse 2 deve ser interpretado como um número propriamente dito; se não houvesse o "#", o 2 seria interpretado como um endereço de memória.)

A segunda instrução do programa, ADC, significa "Adicione com transporte" (ADC = Add with carry). Este comando resulta na adição de dois números *mais* o bit de transporte (se o bit de transporte estiver ativado). Para evitar um resultado impróprio quando da adição de dois números de 8 bits, pode-se usar o comando CLC (Cancelar comando de transporte – CLC = Clear Carry command) antes do comando ADC. Com o bit de transporte cancelado, tudo que a instrução ADC faz é somar 2 e 3.

Assim que a linha "ADC#3" aparece no programa, o 2 que havia sido carregado no acumulador é depositado em uma das entradas da ALU do 6502. E o número 3, com a instrução ADC, é colocado na outra entrada da ALU. A ALU realiza, então, a instrução ADC recebida: soma 2 e 3 e coloca o resultado de volta no acumulador.

Vejamos, agora, a terceira e última instrução do programa ADDNRS. A instrução na linha 3, STA, significa "Armazene o conteúdo do acumulador" (STA = Store the contents of the accumulator) no endereço de memória indicado na instrução. Como o acumulador contém agora o valor 5 (2 mais 3), o número 5 deverá ser armazenado em algum lugar.

Como podemos ver, o endereço de memória que segue a instrução STA é $0300 – o equivalente hexadecimal do número decimal 768. Resulta, então, que o número 5 deverá ser armazenado na localização de memória $0300.

Observemos o número hexadecimal $0300 na linha 3. Como não há o símbolo "#" à frente do número $0300, o assembler Apple ProDOS não irá interpretá-lo como um número, e sim como um endereço de memória, uma vez que não há outras indicações de identificação. (Se desejássemos que o assembler interpretrasse $0300 como um número,

teríamos de escrevê-lo como # $0300. Quando um símbolo "#" e um cifrão aparecem à frente de um número, este é interpretado como um número hexadecimal.)

Se a terceira linha do programa, entretanto, fosse "STA# $0300", isto seria um erro de sintaxe, pois STA (Armazene o conteúdo do acumulador em...) é uma instrução que precisa, necessariamente, ser seguida por um valor que possa ser interpretado como um endereço de memória.

O REGISTRO X

O *Registro X* (abreviado "X") é um registro de 8 bits com uma característica muito especial. Ele pode ser incrementado e decrementado automaticamente, através de um par de instruções de 1 byte muito convenientes: INX, que representa "Incremente o registro X" e DEX, "Decremente o registro X". Dada a facilidade com que é possível incrementar e decrementar o registro X, ele é freqüentemente utilizado como um registro-índice, ou contador, durante *loops* ou instruções do tipo read/data, em programas. Quando as características especiais de incremento e decremento não estão sendo utilizadas, o registro X pode ser usado para armazenamento temporário de dados.

O REGISTRO Y

O *Registro Y* (abreviado "Y") é também um registro de 8 bits, podendo ser incrementado ou decrementado com um par de instruções de 1 byte. Os mnemônicos usados para incrementar e decrementar o registro Y são INY e DEY. Quando não são utilizadas essas características, o registro Y também pode armazenar dados temporariamente.

O CONTADOR DE PROGRAMA

O contador de programa (PC = Program Counter) foi descrito durante a discussão sobre a memória do computador. Recordaremos que o PC é um par de registros de 8 bits projetados para serem utilizados em conjunto, como um registro de 16 bits. Quando o 6502/65C02 está rodando um programa em linguagem de máquina, o contador de programa sempre contém o endereço de 16 bits do próximo registro de memória a ser acessado pelo programa. Quando essa instrução tiver sido realizada, o endereço da próxima instrução é carregado no contador de programa.

Conforme já mencionamos, os dois registros de 8 bits que compõem o contador de programa são, geralmente, chamados de PCL – Program Counter-Low – e PCH – Program Counter-High.

O PONTEIRO DA PILHA (STACK POINTER)

O "stack pointer" (abreviado "S" ou "SP") é um registro de 8 bits que sempre "aponta" ou contém o endereço do elemento que se encontra no topo de um bloco de RAM, chamado "hardware stack". Essa "pilha de hardware; ou simplesmente "pilha", é um bloco especial de memória no qual os dados são, em geral, armazenados temporariamente durante a execução de um programa. Quando são utilizadas sub-rotinas num programa em assembly, o 6502B/65C02 usa a pilha com uma localização de armazenamento temporário dos endereços de retorno. A partir do Capítulo 7, aprenderemos a utilizar a pilha para outras finalidades, em programas em linguagem assembly.

O REGISTRO DE STATUS DO PROCESSADOR

O *processor status register* (também chamado *status register*, mas abreviado como "P", para não confundir com o *stack pointer*) é um registro de 8 bits, que realiza o acompanhamento dos resultados de operações realizadas pelo 6502B/65C02. Estudemos este importante registro:

O registro P é construído de forma diferente, com relação aos outros registros do 6502B/65C02, e utilizado de forma diferente também. Ao contrário dos demais, ele não foi projetado para armazenar ou processar números de 8 bits. Os bits do registro P são usados como indicadores (flags), projetados para manter controle sobre vários tipos de informações importantes.

Quatro de seus 8 bits são chamados *indicadores de status (status flags)*, que são, com as respectivas abreviaturas:

- indicador de transporte – C (carry flag);
- indicador de overflow – V;
- indicador de negativo – N;
- indicador de zero –Z.

Estes indicadores são utilizados para manter o acompanhamento dos resultados de operações realizadas pelos outros registros internos do 65C02/6502B.

Como o registro P é um registro de 8 bits, sobram ainda 4 indicadores: três deles são os chamados *indicadores de condição (condition flags)*, utilizados para determinar se certas condições existem em um programa:

- indicador de desativamento de interrupção – I (Interrupt disable flag);
- indicador de "break" – B (Break flag);
- indicador de modo decimal – D

O oitavo bit do registro P não é utilizado.

CONFIGURAÇÃO DO REGISTRO P

O registro de status do processador pode ser visualizado como uma caixa retangular dividida em oito compartimentos, cada um contendo um dos 8 bits do registro P. Neste registro, cada bit é utilizado como um indicador ("flag").

Se um determinado bit apresenta o valor binário 1, diz-se que ele está ativado (*set*). Se apresenta o valor binário 0, diz-se que está "zerado" ou "desativado" (clear).

Os bits, no registro de status do 65C02/6502B – como os bits em quaisquer registros de 8 bits – são normalmente numerados de 0 a 7. Por convenção, o bit mais à direita é, geralmente, designado como o bit 0, e o mais à esquerda, como o bit 7. As posições destes bits no registro de status do 6502B/65C02 estão ilustradas na Figura 3.2.

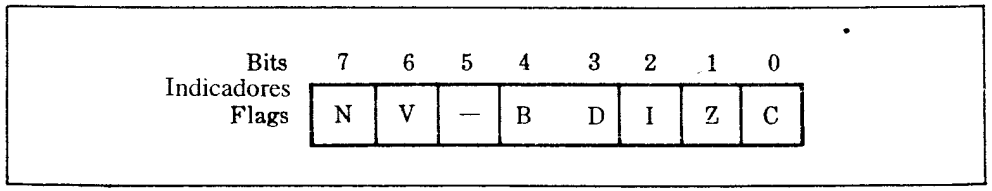


Figura 3.2 Registro de status do processador no 65C02/6502B.

A seguir, apresentamos a lista completa e as explicações sobre os indicadores no registro de status do 65C02/6502B.

Bit 0: O indicador de transporte (C = Carry Flag)

O indicador de transporte é um dos bits mais atarefados no registro de status do 65C02/6502B. Ele diz se um número deve ser transportado de um byte a outro, durante uma operação aritmética.

Num processador de 8 bits, como o 65C02/6502B, o indicador de transporte é de especial importância. Sem ele, o 65C02/6502B seria incapaz de realizar operações com números maiores que 255. Se é necessário realizar operações com números maiores que 255, ou se o resultado de uma operação for maior que 255, esse número precisa ser desmembrado em segmentos de 8 bits, para ser processado e, então, recomposto novamente. O indicador de transporte do registro P desempenha um papel importante neste "corta e cola" matemático.

Se dois números de 16 bits precisam ser somados, cada um deles, primeiramente, tem de ser "quebrado" em 2 bytes de 8 bits. Os bytes de ordem inferior dos dois números são, então, somados. Se esta soma apresentar um resultado superior a 255, o indicador de transporte, automaticamente, será ativado. Então, se os bits de ordem superior dos dois números forem somados, o bit de transporte será acrescentado automaticamente a essa soma.

O indicador de transporte é ativado e zerado não somente nas operações de adição, mas também em várias outras operações realizadas pelo 65C02/6502B. Instruções detalhadas sobre a utilização do indicador de transporte serão apresentadas nos Capítulos 9 e 10.

A instrução em linguagem assembly para "zerar" o indicador de transporte é CLC ("Clear carry"). O mnemônico que o ativa é SEC ("Set carry").

Bit 1: O Indicador de Zero (Z = Zero Flag)

O indicador de zero é ativado quando o resultado de uma operação aritmética, lógica ou de comparação é 0. Quando um endereço de memória ou um registro índice for decrementado até 0, isto também ativará o indicador de zero.

Ao escrever rotinas que fazem uso do indicador de zero, lembre-se desta convenção do 6502, que pode parecer um tanto estranha: quando o resultado de uma operação é zero, o indicador de zero é *ativado* (para 1), e quando o resultado de uma operação é *diferente de zero*, o indicador de zero é *desativado* (para 0).

Não existem instruções em assembly para ativar ou desativar o indicador de zero, uma vez que este é um bit exclusivamente de leitura (read only bit).

Bit 2: O Indicador de Desativamento de Interrupção (I = Interrupt Disable Flag)

Na terminologia da linguagem assembly, uma instrução de *interrupção* (interrupt) provoca uma parada em todas as operações do computador, para que alguma operação muito importante, ou crítica em termos de tempo, possa ser realizada. Algumas interrupções são chamadas *interrupções mascaráveis* (maskable interrupts), pois podem ser evitadas através de instruções especiais. Outras são chamadas *não-mascaráveis* (nonmaskable), pois não existe nenhuma forma pela qual o programador possa impedi-las de ocorrer. Estas, entretanto, não são usadas na programação do Apple IIc/IIe.

Podemos permitir uma interrupção mascarável, "zerando" o bit de desativamento de interrupção, no registro de status do processador. Quando o indicador está ativado, as interrupções mascaráveis não podem ocorrer. Quando está desativado podem.

O mnemônico que desativa o indicador de interrupção é CLI (Clear Interrupt), e o que o ativa é SEI (Set Interrupt).

Bit 3: O Indicador de Modo Decimal (D = Decimal Mode Flag)

Normalmente, o processador 65C02/6502B opera no chamado *modo binário*, usando números binários "padrão" de 8 bits. Mas sua CPU pode também trabalhar no que é conhecido como *decimal codificado em binário*, ou modo BCD (Binary-coded decimal). Para isto, é necessário ativar o indicador de modo decimal no registro de status do 65C02/6502B.

Ao trabalhar em modo decimal, o 65C02/6502B utiliza somente os 10 dígitos decimais – os números de 0 a 9. Os dígitos hexadecimais, A a F, não são utilizados em operações BCD. Além disso, cada dígito, em um número BCD, é tratado como um byte individual. Por exemplo, são necessários 3 bytes (um por dígito) para expressar o número decimal 255 como um número BCD. Na memória do computador, o número 255 seria armazenado assim:

Número BCD	2	5	5
Equivalente binário	00000010	00000101	00000101

Isto é bem diferente, é claro, da maneira pela qual o número 255 seria expresso em notação binária (não BCD) convencional. Em aritmética binária, a que mais se usa em programas em assembly, o número decimal 255 seria expresso como um número hexadecimal.

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

Como se pode ver, à taxa de um byte por dígito, é utilizada uma quantidade muito maior de memória do que a necessária para armazenar números binários convencionais. (É possível "comprimir" dígitos BCD em metade do espaço necessário, com procedimentos especiais e um tempo maior de processamento, como veremos em capítulos subseqüentes.)

Outra desvantagem da aritmética BCD é o fato de ser mais lenta do que a binária; porém, a matemática BCD é baseada no número 10, não no número 8, de forma que é mais acurada que a matemática hexadecimal, em termos dos problemas reais, em aritmética base 10.

Outra vantagem dos números BCD é que são convertidos mais facilmente em números decimais que os binários. Os números BCD são, desta forma, utilizados algumas vezes em programas que pedem uma imediata apresentação de números na tela do monitor.

Os números BCD serão discutidos mais detalhadamente num capítulo mais adiante. Por enquanto, basta lembrar que, quando o indicador de modo decimal do registro de status está ativado, o 65C02/6502B realizará toda a aritmética com números BCD.

Se você não estiver utilizando aritmética BCD em seus programas em assembly, deve certificar-se de que o indicador de modo decimal esteja desativado antes que o computador inicie a realização de operações aritméticas.

A instrução, em linguagem assembly, que desativa o indicador de modo decimal é CLD. A instrução que o ativa é SED.

Bit 4: O indicador de Break (B = Break Flag)

O indicador de break é geralmente usado pelos programadores durante a depuração ("debugging") de programas em linguagem assembly. Ele é ativado pela instrução BRK, utilizada quando da depuração de programas.

Quando um programador está escrevendo um programa em linguagem assembly, as instruções BRK são geralmente inseridas em pontos críticos do programa. O indicador de break é ativado seguindo uma instrução BREAK do software.

Quando encontradas durante o processamento do programa, certas operações que levam a indicações de erro retornam o controle do computador ao programador.

Assim, a instrução BRK pára a execução do programa e faz com que o conteúdo dos registros A, X, Y, P e S do 65C02/6502B seja apresentado na tela, onde eles poderão então ser examinados para determinar o que ocorreu durante o processamento do programa.

Uma vez depurado um programa, todas as instruções BRK colocadas com a finalidade de auxiliar a depuração são usualmente removidas, e o programa rodará, então, normalmente.

Bit 5: Não-Utilizado

Os projetistas do 6502 deixaram este bit sem utilização. Como veremos mais adiante neste capítulo, entretanto, o bit 5 do registro P é utilizado nos dois novos primos de 16 bits do 6502, o 65802 e o 65816.

Bit 6: O Indicador de Overflow (V = Overflow Flag)

Este indicador é usado para detectar o overflow, em um número binário, do bit 6 para o bit 7. É utilizado, principalmente, em operações que lidam com números com sinais (de mais e menos). Em um número binário comum, sem sinal, o bit mais alto, ou mais significativo, é o bit 7. Em um número com sinal, entretanto, o bit 7 é usado para indicar esse sinal. Se o bit 7 de um número com sinal está desativado, o número é positivo; se o bit 7 está ativado, o número é negativo.

Uma vez que o bit 7 de um número com sinal não é usado como parte integrante do próprio número, o bit mais significativo de um número com sinal é o bit 6. É difícil, portanto, realizar operações de transporte e empréstimo em problemas aritméticos que lidam com números com sinais, de forma que o indicador de overflow é utilizado para manter controle das operações de transporte e empréstimo em aritmética binária com sinais.

Num problema de adição, o indicador de overflow é ativado quando o bit 7 de ambos os membros da adição têm o mesmo valor, e o bit 7 da soma será o valor oposto. Numa operação de subtração, o indicador de overflow é ativado quando o bit 7 do subtraendo e do minuendo são opostos, e o bit 7 do resultado apresentará o mesmo valor do bit 7 do subtraendo. Em outras palavras, o indicador de overflow é ativado quando existe um overflow do bit 6 para o bit 7, mas não existe transporte externo, ou, inversamente, quando não há overflow do bit 6 para o bit 7, mas existe transporte externo.

Uma forma mais técnica de expressão seria a de dizer que o indicador de transbordamento é ativado pela execução de uma operação OU exclusivo no bit de "transporte do" e "transporte para" que é o bit 7. Este procedimento pode parecer um tanto complicado, mas, sem a ajuda do indicador de overflow, seria impossível escrever programas para a execução de operações que envolvessem números com sinais.

O mnemônico, em linguagem assembly, que desativa o indicador de transbordamento é CLV. O indicador V pode ser desativado, mas não existe uma instrução específica para ativá-lo.

Bit 7: O Indicador de Negativo (N = Negative Flag)

O indicador de negativo é ativado quando o resultado de uma operação é negativo, e desativado quando o resultado for 0 ou positivo. O indicador de negativo é, freqüentemente, usado em operações de comparação e em contagens regressivas em "loops" até 0.

O bit N é um bit somente de leitura, de forma que não existem instruções para ativar ou desativar o mesmo.

ARQUITETURA DO 65802/65816

Como ilustrado na Figura 3.1, a arquitetura dos microprocessadores 6502, 65C02 e 6502B pode ser representada pelo mesmo diagrama de blocos. Há diferenças significativas, porém, entre a arquitetura do 65C02/6502B e dos dois novos microprocessadores de 16 bits da família 6502, o 65802 e o 65816.

O INDICADOR E (EMULAÇÃO)

Tanto no 65802 como no 65816 existe um indicador de status extra, chamado indicador de emulação (emulation flag). Este indicador não faz parte do registro de status do processador, mas é uma chave liga-desliga que constitui parte integrante da própria CPU. Entretanto, a instrução XCE pode ser usada para mudar o estado do indicador E. Utilizando-se uma instrução XCE durante um programa em assembly, pode-se, temporariamente, intercambiar as posições dos indicadores flutuantes E e C (este do registro de status do processador). Como o indicador de transporte C (carry flag) ordinariamente reside no bit 0 do registro P, o efeito da instrução XCE é o de armazenar o conteúdo do registro E no bit 0 do registro P, enquanto coloca temporariamente o indicador de transporte no registro E. Ocorrida esta troca, uma instrução em linguagem assembly como SEC (Set carry = ativar transporte) ou CLC (Clear carry = desativar transporte) pode ser usada para alterar o estado do indicador E. Outra instrução XCE pode, então, ser emitida, para restaurar as posições originais dos indicadores E e C. O estado do indicador E será, então, mudado, e o indicador C poderá ser novamente utilizado de forma normal.

Quando o indicador E do registro P é desativado para 0, seu valor default, o 65802/65816 é um microprocessador de 16 bits. Quando, entretanto, o indicador E é ativado para 1, o 65802/65816 é colocado no chamado modo de emulação do 6502.

Estando em seu modo default de 16 bits, o 65802/65816 pode ser programado em uma linguagem que é uma evolução da linguagem assembly normal do 6502. Quando no modo de emulação do 6502, pode ser utilizado exatamente como um 6502 padrão (ou como um 65C02 ou um 6502B). Neste modo, o 65802/65816 pode reconhecer e processar as mesmas instruções em linguagem de máquina que seus primos de 8 bits, de forma que pode rodar todo o software escrito para o 6502, o 6502B e o 65C02.

INDICADORES ADICIONAIS DO REGISTRO P

Existem, ainda, mais duas diferenças entre o registro de status do processador dos *chips* 6502B/65C02 e o registro P do 65802 e do 65816. O bit 4 do registro P – o indicador de break (B) no 6502B/65C02 – é conhecido como *indicador X* ou *indicador de seleção do registro de índice (index register select flag)* no 65802/65816.

O bit 5 do registro P, que não é utilizado no 6502B/65C02, é chamado *indicador M* ou *indicador de seleção de memória* (memory select flag), no 65802/65816. A Figura 3.3 mostra como são codificados os bits no registro P do 65802/65816 quando no modo 16 bits.

O Indicador M

Quando ambos os indicadores, E e M, no 65802/65816, estão desativados, o acumulador do *chip* é de 16 bits e é chamado de *registro C*. Quando o acumulador está neste modo, seus 8 bits inferiores são chamados de *registro A* e seus 8 bits superiores de *registro B*. Porém, os nomes das instruções em assembly para o acumulador não são afetados por essa mudança: LDA permanece LDA e STA permanece STA.

Há algumas alterações, é claro, na maneira pela qual LDA, STA e outras instruções para o acumulador trabalham quando o 65802/65816 se encontra no modo 16 bits. Estando neste modo, instruções como LDA e STA podem ser utilizadas com operandos de 2 bytes. Quando um operando de 2 bytes é usado desta maneira, seu byte inferior afeta o registro A do acumulador, e seu byte superior o registro B. Por exemplo, o comando

```
LDA $ 8000
```

carregará o valor numérico 00 no registro A e o valor numérico 80 no registro B.

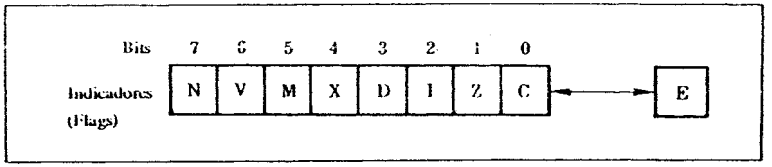


Figura 3.3 O registro de status do processador no 65802/65816 (quando no modo 16 bits).

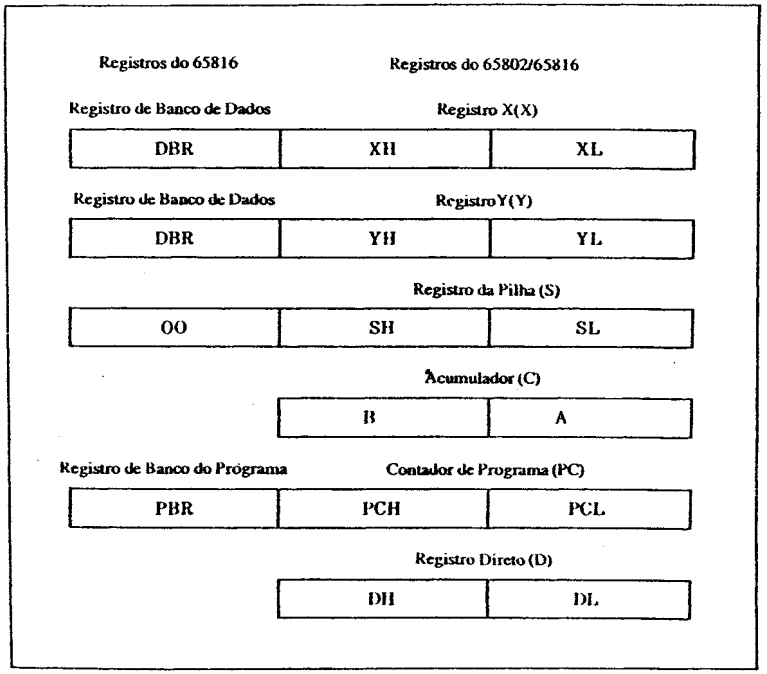


Figura 3.4 Registros internos dos microprocessadores 65802 e 65816.

Entretanto, o comando

```
LDA $800
```

carregará o registro A com o valor do registro de memória $8000 e o registro B com o valor do registro de memória $8001.

O Indicador X

Quando ambos os registros, E e X do 65802/65816 estão desativados, os registros X e Y são de 16 bits. Quando os registros X e Y estão nesse modo, seus 8 bits inferiores são conhecidos como XL e YL, e seus 8 bits superiores como XH e YH.

O REGISTRO S

Quando o registro E do 65802/65816 está desativado, o registro da pilha (S = Stack register) é de 16 bits. Esta expansão torna possível a colocação da pilha de hardware em qualquer lugar da memória.

O REGISTRO D

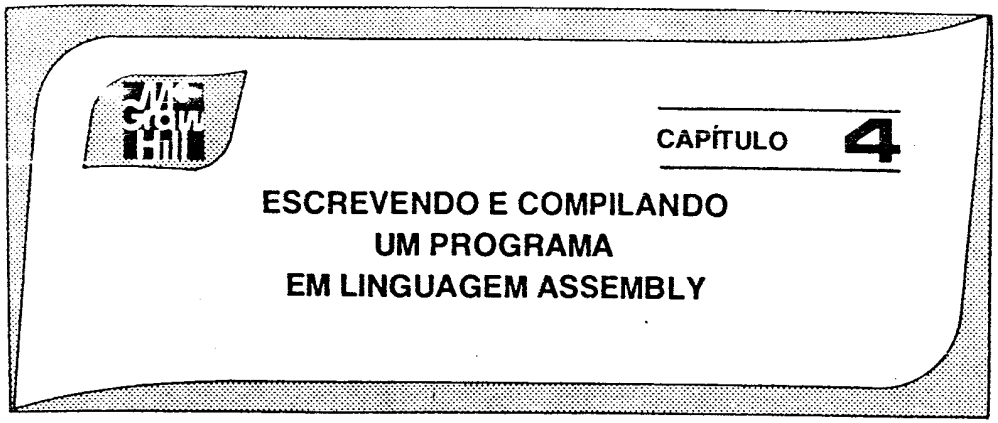
O 65802/65816 possui mais um registro interno do que seus parentes de 8 bits. Este registro extra é chamado de *registro direto da página* ou *registro D*. Quando o registro E do 65802/65816 está desativado, o registro D pode ser utilizado para aumentar a capacidade do *chip* em manusear um tipo rápido e útil de endereçamento, chamado endereçamento da página zero (zero-page addressing) na linguagem assembly 6502. A operação do registro D será abordada em detalhes no Capítulo 7.

OS REGISTROS DE 24 BITS NO 65816

Além do registro D, o 65816 possui outros dois registros especiais: um *registro de banco do programa* ou PBR (program bank register) e um *registro de banco de dados* ou DBR (data bank register).

O registro de banco do programa é um registro de 8 bits que pode estender o comprimento do contador de programa de 65816 para 24 bits. Quando o PRB é usado, o 65816 pode endereçar até 16 megabytes de memória.

Com a ajuda dos registros de banco de dados, os registros X e Y do 65816 podem, também, ser expandidos para 24 bits. Esta capacidade provê o 65816 de um método simples para endereçar até 24 megabytes de dados. A capacidade de endereçamento do 65816 será explicada com maiores detalhes no Capítulo 7. Por enquanto, a Figura 3.4 é um diagrama de blocos que mostra os registros internos do 65802 e do 65816.



CAPÍTULO 4

ESCREVENDO E COMPILANDO UM PROGRAMA EM LINGUAGEM ASSEMBLY

Neste capítulo, finalmente temos a oportunidade de escrever um programa real em linguagem assembly. Para isto, é claro, precisamos ter um pacote de software chamado assembler para linguagem de máquna. Devemos nos lembrar de que, na Introdução, foram mencionados os três assemblers utilizados para escrever os programas deste livro. Para tirar o máximo proveito do livro, deveremos possuir pelo menos um deles:

- o pacote *Apple ProDOS Assembler Tools*, da Apple;
- o assembler *Merlin Pro*, da Roger Wagner Publishing, Inc., de Santee, California;
- o assembler *ORCA/M*, da The Byte Works, Inc., de Albuquerque, New Mexico.

Estes três assemblers foram comparados de forma bem abrangente na Introdução; agora podemos começar a usar o assembler que escolhemos.

Este capítulo é dividido em três partes. Primeiramente, será explicado como digitar e montar um programa utilizando o assembler Apple ProDOS, depois utilizando o Merlin Pro e, finalmente, o ORCA/M.

Não importando o assembler com o qual formos trabalhar, deveremos ler cuidadosamente a discussão sobre o Apple ProDOS, uma vez que somente ali será explicado o funcionamento do programa, linha por linha.

O termo "assembler" pode ter dois significados diferentes, dependendo da forma como é usado. No sentido mais preciso, um assembler é somente uma parte de um pacote

de software editor/assembler: a parte que realmente realiza o trabalho de conversão da linguagem assembly em linguagem de máquina. Algumas vezes, porém, o termo se refere a um kit completo de desenvolvimento de software, como assembler ProDOS, ORCA/M ou Merlin Pro. Este livro utiliza o termo em ambos os significados; o contexto no qual o termo aparece deverá tornar claro o sentido.

O ASSEMBLER APPLE ProDOS

O *Apple ProDOS Assembler Tools* é composto por quatro diferentes programas:

- Um editor, que pode ser usado para criar e editar programas em linguagem assembly e gravá-los em disco (e também para criar e editar arquivos EXEC em ProDOS e programas em BASIC).
- Um assembler, usado para converter arquivos código-fonte em programas executáveis em linguagem de máquina.
- Um utilitário de depuração assembly, chamado *Bugbyter*, que permite localizar e corrigir erros em programas em assembly.
- Um carregador/relocador, usado para carregar e executar programas em assembly durante a execução de programas em BASIC.

Todos os programas do pacote Apple ProDOS assembler estão contidos em um único disco. Esse disco não tem proteção contra cópias de forma que podemos (e devemos) obter pelo menos um cópia (backup) antes de começar a utilizar o original para escrever programas, ou usar uma cópia e guardar o original em local seguro.

Carreguemos, então, o disco (boot) da mesma forma utilizada com qualquer outro disco ProDOS. A tela deverá apresentar algo como:

```
PRODOS BASIC 1.0
COPYRIGHT APPLE, 1983
]
```

Após o prompt "]" haverá um cursor piscando. Digite então a linha

```
]-EDASM.SYSTEM
```


O disco girará e será apresentada na tela a seguinte mensagem:

```
PRODOS EDITOR-ASSEMBLER //

ENTER THE DATE AND
PRESS RETURN

DD-MMM-YY
```

Haverá um cursor piscando sobre o primeiro D na última linha da mensagem. Digitando a data sobre as letras DD (data), MMM (mês) e Y Y (ano), veremos então

```
1
PRODOS EDITOR-ASSEMBLER //
BY JOHN ARKLEY
(C) COPYRIGHT 1982

APPLE COMPUTER INC.
@


          DD-MMM-YY

          :
```

(DD-MMM-YY irá apresentar a data digitada).

Ao aparecer esta tela, o assembler estará no modo de Comando, no qual se escrevem e editam programas em assembly.

O programa que escreveremos neste capítulo é o ADDNRS, apresentado no Capítulo 3. O ADDNRS é um programa pequeno e simples: ele simplesmente soma os números 2 e 3 e armazena o resultado em um determinado registro de memória (especificamente, o registro $0300).

A versão do ADDNRS que será escrita com a utilização do Apple ProDOS assembler é chamada ADDNRS.SRC.

INTRODUZINDO O PROGRAMA ADDNRS.SRC

Próximo à parte inferior da tela, logo abaixo da data, há um sinal de dois pontos seguido por um cursor piscante. O prompt ":" é o que sempre se vê quando o assembler

está no modo de comando (ou de edição) – isto é, quando se está utilizando o módulo editor do pacote *Apple ProDOS Assembler Tools*.

Ao ver o prompt, digite A ("Append"). Então aparecerá o número 1 na tela. Esse 1 é um número de linha – o número da primeira linha que você digitará quando iniciar um programa em código-fonte.

Números de Linha em Programas com o Apple ProDOS Assembler

O número 1 apareceu automaticamente na tela; isto se deve ao fato do editor de código-fonte no Apple ProDOS gerar automaticamente *números de linha* em programas código-fonte, começando em 1 e incrementando unitariamente. É importante notar, entretanto, que os números de linha gerados pelo editor Apple ProDOS diferem bastante dos números comumente usados nos programas em BASIC. Diferentemente do BASIC, a linguagem assembly do 6502 não requer o uso de números de linha. De fato, alguns assemblers da família 6502 – como por exemplo, o ORCA/M – não utilizam números de linha.

Os números de linha são opcionais nos programas em assembly, uma vez que nunca iremos nos referir a uma rotina através de seu número na linguagem assembly 6502; segmentos de código-fonte são usualmente acessados com o auxílio de rótulos descritivos (labels). Um rótulo pode ser associado à primeira linha de qualquer rotina em um programa em assembly, e usado para acessar aquela rotina, sempre que for desejado.

Uma vez que os números de linha não são essenciais para a operação dos programas em linguagem assembly, o editor de código-fonte do Apple ProDOS gera o que é algumas vezes chamado de *números de linha relativos*. Um número de linha relativo não é parte integrante de um programa em assembly; ele é colocado somente por conveniência, de forma a facilitar a localização no programa, para programadores e usuários.

Há vantagens e desvantagens na utilização dos números de linha relativos. Nunca precisaremos nos preocupar em numerar ou renumerar as linhas em um programa: os números de linha relativos são gerados e incrementados automaticamente. Quando o editor numera as linhas num programa, jamais pula uma linha. E também renumerará automaticamente as linhas, sempre que uma delas for inserida ou eliminada.

É importante tratar com muito cuidado os números de linha relativos ao escrever e editar um programa, pois eles podem se alterar sem aviso. Ao efetuar a eliminação de uma seqüência de linhas, por exemplo, é melhor iniciar eliminando a linha de número mais alto, pois a eliminação de uma linha provoca a alteração na numeração das linhas subseqüentes, o que poderia levar a eliminação inadvertida de uma ou mais linhas.

O número 1, em sua tela, é seguido por um espaço e um cursor piscante. Digite agora mais um espaço e, sem tocar a tecla RETURN, digite

```
ORG $1000
```

A linha 1 de seu programa será

```
1  ORG $1000
```

Agora tecle RETURN. O editor ProDOS passará para a próxima linha, apresentando ali um 2, o que indica que está pronto para receber a linha 2 do programa ADDNRS.SRC.

Após o número de linha relativo 2 – mas, desta vez, sem um espaço extra – digite um ponto-e-vírgula. A linha 2 de seu programa será

```
2 ;
```

Pressione RETURN. O editor avançará para a linha 3. Digite, a seguir, estas duas linhas:

```
3 ; ADDNRS.SRC
4 ;
```

Tecle RETURN, e você verá

```
1  ORG $1000
2 ;
3 ; ADDNRS.SRC
4 ;
5
```

Breve aprenderemos o significado destas linhas. Antes, porém, vamos examinar algumas das características mais importantes do sistema assembler/editor do Apple ProDOS:

Funções de Edição de um Único Caractere

Se cometermos um erro ao digitar uma linha de código, há várias maneiras de corrigi-lo. Se estamos no modo "append", podemos mudar qualquer letra em uma linha, simplesmente digitando outra letra sobre aquela. Podemos eliminar o caractere que se encontra sob o cursor, digitando **CONTROL-D**. Podemos eliminar o caractere à esquerda do cursor, usando a tecla DELETE. Podemos inserir um caractere em uma linha, na posição do cursor, digitando **CONTROL-I**. Se mudarmos de idéia a respeito da correção

de uma linha e resolvermos restaurar a versão original, é só digitar **CONTROL-R.** (Uma palavra de precaução, entretanto: quando o assembler ProDOS se encontra no modo "edit", algumas das respostas aos comandos de edição são diferentes das respostas aos mesmos comandos quando no modo "add". Por exemplo, no modo "edit", a tecla DELETE apaga o caractere sob o cursor, não o caractere à esquerda.)

Funções de Edição de Linhas

O editor de código-fonte do Apple ProDOS é um editor orientado a linhas; podemos nos movimentar em uma linha utilizando as teclas SET ESQUERDA e DIREITA, mas não podemos passar de uma linha a outra usando as teclas SETA PARA CIMA e SETA PARA BAIXO. Ao desejar sair de uma linha, devemos usar a tecla RETURN. E cada vez que teclarmos RETURN ao final de uma linha, ela será automaticamente incluída no programa.

Quando terminarmos a digitação de um programa – ou quando desejarmos parar para voltar e efetuar alguma edição – podemos fazer com que o assembler pare de gerar novos números de linha, simplesmente teclando RETURN duas vezes seguidas. O editor parará a geração de números de linha, e o prompt ":" aparecerá na tela.

```
14-JUN-85
:
```

Poderemos, então, teclar **A** ("append"), exatamente como no início da sessão de programação. O assembler reapresentará a numeração como estava quando da parada, e poderemos continuar a escrever o programa.

Dois outros comandos que podem ser usados após o prompt ":" são L (LIST) e P (PRINT). Se teclarmos um comando L, o programa será listado, completo, com os números de linha, no monitor. O comando P também apresentará a listagem no monitor, mas sem os números de linha.

Se desejarmos ver uma linha específica, poderemos digitar o número dessa linha após os comandos L ou P. Se desejarmos ver um bloco de linhas, poderemos especificar o primeiro e o último número de linha desejados, usando este formato:

início # – fim #

ou

início # – fim #

O Comando DELETE

Outro comando que pode seguir o prompt ":" é D(DELETE). Para usar o comando DELETE, é só teclar **D** seguido do número da linha (ou linhas) que deseja eliminar. Suponhamos, por exemplo, que desejamos eliminar as linhas 2 e 3 na listagem acima. Para tanto, basta digitar **D2-3** após o prompt ":".

Outro comando, ainda, que pode ser usado após o promtp ":" é o comando I(INSERT). Para usá-lo, devemos teclar **I** após o prompt, seguido do número de linha onde desejamos que a nova linha seja inserida. Vamos inserir outro ponto-e-vírgula na linha 2 do programa que estamos criando. Devemos teclar RETURN, para obter um prompt dois-pontos, e digitar

```
I2
```

O assembler editor responderá com o número 2.

Teclando, agora, um ponto-e-vírgula seguido de dois RETURNs, o editor apresentará seu prompt ":" novamente, e poderemos teclar **L** para obter uma listagem. Foi inserida uma nova linha contendo um ponto-e-vírgula, na linha 2 do programa. E que, conforme já foi explicado, o assembler renumerou automaticamente as demais linhas.

Agora que já vimos o sistema de renumeração de linhas em ação, vamos eliminar o ponto-e-vírgula extra que foi adicionado ao programa. Tecle RETURN, para obter um prompt dois-pontos, e digite **D2**. Após isso, tecle **L** para gerar uma nova listagem do programa.

```
1  ORG $1000
2  ;
3  ; ADDNRS.SRC
4  ;
```

Além dos comandos A, I e D que você acabou de utilizar, existe também o comando R, que pode ser utilizado para restaurar uma linha. O editor de código-fonte do Apple ProDOS apresenta ainda comandos para copiar linhas, localizar e substituir seqüências (strings) e realizar várias outras funções úteis. Detalhes de como utilizar essas funções de edição são fornecidos pelo manual que acompanha o Apple ProDOS assembler.

Outras Características do Editor ProDOs

O Editor do Apple PRoDOS possui várias outras características, que não serão explicadas neste capítulo. Por exemplo, há um comando, SWAP, que permite esconder um

programa código-fonte em um bloco de memória, para que se possa escrever ou editar outro programa. Podemos, então, intercambiar (SWAP) os programas, até estarmos prontos para gravá-los (SAVE) ou eliminar um deles, ou ambos, utilizando um outro comando do assembler, KILL2.

COMPLETANDO O PROGRAMA ADDNRS

Vamos, agora, terminar a digitação do programa ADDNRS. O Programa 4.1 é a listagem completa, em código-fonte, desse programa, e agora é um bom momento de introduzir o restante da listagem no computador. Antes de iniciar a digitação, entretanto, um alerta: o assembler ProDOS, assim como a maioria dos assemblers, é rígido com relação a espaços.

Assim, devemos nos certificar que o programa seja digitado exatamente como no Programa 4.1. Nas linhas de 2 a 5, não deverão existir espaços extra entre o número de linha relativo gerado pelo assembler e as instruções que vêm a seguir. (O editor ProDOS insere automaticamente um espaço, após o número de linha.) Deve haver um espaço extra após a linha 1, e também devem existir espaços extra nas linhas de 6 a 10.

Se respeitarmos as regras de espaçamento (explicadas detalhadamente mais adiante, neste capítulo), a listagem do ADDNRS.SRC deverá apresentar-se exatamente como a do Programa 4.1.

Programa 4.1
O PROGRAMA ADDNRS.SRC
(Versão Apple ProDOS Assembler)

```
 1  ORG $1000
 2 ;
 3 ; ADDNRS.SRC
 4 ;
 5 ADDNRS CLD
 6  CLC
 7  LDA #2
 8  ADC #3
 9  STA $0300
10  RTS
```


LISTANDO O PROGRAMA

Terminando de digitar a linha 10 do programa, tecle RETURN duas vezes. Digite, então, **L** ou **LIST**, e o programa completo será listado na tela.

Você deve ter notado que o programa ADDNRS.SRC que digitou e listou é o mesmo do Capítulo 3. Mais tarde, ao rodar o programa, aprenderá como ele funciona. Você já sabe que ele adiciona os números 2 e 3 e guarda o resultado no endereço de memória $0300 – ou, em notação decimal, no registro de memória 768. Você pode ver, no programa, que os números são somados nas linhas 7 e 8, e que a soma é armazenada no registro $0300 na linha 9.

ESPAÇAMENTO

O Programa 4.2 é um "diagrama explodido" do programa ADDNRS.SRC. Ele não segue as regras de espaçamento citadas há alguns parágrafos, mas esclarece como as informações contidas em uma listagem código-fonte podem ser espalhadas em *campos* ou colunas.

Programa 4.2
O PROGRAMA ADDNRS.SRC, COLUNA POR COLUNA

NÚMERO DE LINHA	RÓTULOS E LEMBRETES	CÓDIGO OPERACIONAL	OPERANDO	COMENTÁRIOS
1		ORG	$1000	
2	;			
3	; ADDNRS.SRC			
4	;			
5	ADDNRS	CLD		
6		CLC		
7		LDA	#2	
8		ADC	#3	
9		STA	$0300	
10		RTS		

CAMPOS NAS LISTAGENS CÓDIGO-FONTE

Pode-se notar, observando o Programa 4.2, que os programas em linguagem Assembly criados com o assembler/editor Apple ProDOS podem ser divididos em várias

colunas ou campos. Os campos ilustrados no Programa 4.2 são os campos *rótulos, código operacional, operando e comentários*. Além deles, uma coluna é utilizada para os números de linha. Porém, como os números de linha são opcionais em linguagem assembly, essa coluna não é considerada como um campo separado.

O Campo Rótulos

Rótulos, quando usados, sempre ocupam o primeiro campo em uma listagem de código-fonte em assembly. A maioria dos comandos em assembly pode ser identificada com títulos, e estes são realmente necessários em alguns tipos de comando em linguagem assembly.

Embora opcionais, os rótulos são muito importantes na linguagem assembly 65C02/6502B, uma vez que são utilizados, em lugar dos números de linha, para endereçar rotinas e sub-rotinas. No programa ADDNRS.SRC, a abreviatura ADDNRS da linha 5 é um título; assim, aparece no primeiro campo do Programa 4.2.

Num programa em assembly, lembretes podem começar, também, no campo rótulo. Porém, devem ser precedidos por algum tipo de identificador, usualmente um asterisco ou ponto-e-vírgula (dependendo do assembler). No caso do Apple ProDOS, pode ser usado tanto um asterisco quanto um ponto-e-vírgula.

Quando um lembrete inicia no campo rótulo ele pode estender-se através dos outros campos. Um lembrete, como qualquer outra linha num programa em assembly, sempre termina com um RETURN.

Quando um rótulo é designado a uma rotina ou programa em assembly, o programa pode, então, ser gravado em disco e, mais tarde, ser usado como uma sub-rotina ou rotina secundária em um programa maior. Uma vez que assinalamos o programa ADDNRS.SRC com um rótulo – o rótulo ADDNRS – que aparece na linha 5, esse programa pode ser incluído em um programa maior, de onde será acessado com a instrução JSR ADDNRS (o equivalente, em assembly, da instrução GOSUB em BASIC) ou com a instrução JMP ADDNRS (que funciona como a instrução GOTO do BASIC). Outras instruções de *jumping* (pular) e *branching* (ramificar) podem ser usadas com rotinas e sub-rotinas com rótulos; isto será objeto de capítulos posteriores.

Se o ADDNRS.SRC fosse usado como sub-rotina em um programa maior, a instrução RTS (retorne da sub-rotina) da linha 10 terminaria a sub-rotina e devolveria o controle ao programa principal. Se ADDNRS.SRC fosse rótulo de um programa completo, a instrução RTS representaria o fim do programa. (As instruções JSR e JMP serão discutidas em capítulos subseqüentes.)

Um rótulo pode ser constituído de apenas um caractere ou apresentar o comprimento que um comando permite. A maioria dos programadores usa rótulos com 3 a 6 caracteres de extensão. Entretanto, alguns títulos (como A, X, Y) apresentam restrições, e títulos muito longos são indesejáveis, pois reduzem a velocidade do programa.

O Campo Código Operacional

Um mnemônico de *código operacional* (op-code) é um outro nome para uma instrução em assembly. Há 57 mnemônicos no microprocessador 6502B usado no Apple IIe e mais dez mnemônicos adicionais no jogo de instruções expandido do 65C02 usado no Apple IIc.

Mnemônicos, como CLC, CLD, LDA, ADC, STA e RTS, são digitados no campo Código Operacional das listagens código-fonte em assembly. Quando escrevemos um programa usando o Apple ProDOS Assembler Tools, cada mnemônico utilizado deve começar pelo menos dois espaços após o número de linha, ou um espaço após um rótulo. Um mnemônico colocado em campo errado não será assinalado como erro no ato da digitação do programa, mas o será quando o programa for compilado (assembled).

O campo op-code é também usado para *diretivas* e *pseudo-ops* – palavras e símbolos que são introduzidos no programa como mnemônicos, mas não fazem parte do jogo de instruções do 6502. A diferença entre um op-code e um pseudo-op é que um op-code diz ao microprocessador para executar algo, enquanto um pseudo-op diz ao assembler para executar algo durante a compilação.

Assim, embora pseudo-ops e op-codes sejam parecidos, na realidade são muito diferentes. Como os op-codes fazem parte do jogo de instruções do 6502, eles nunca variam de assembler para assembler. Mas os pseudo-ops variam, embora haja uma certa padronização.

A diretiva ORG na linha 1 do ADDNRS.SRC é um exemplo de pseudo-op. Quase todos os assemblers usam uma diretiva de origem de um tipo ou de outro, mas alguns usam um símbolo de igual (=) em lugar da abreviatura ORG. Em um programa em linguagem assembly 6502/65C02, a diretiva ORG (ou seu equivalente) é usada para dizer ao assembler onde o programa em assembly será armazenado na memória após sua conversão em linguagem de máquina.

O Apple ProDOS assembler reconhece uma variedade de pseudo-ops; alguns serão vistos neste capítulo, e todos estão detalhados no manual de instruções do ProDOS Assembler Tools.

O Campo Operandos

O campo Operandos em uma listagem código-fonte Apple IIc/IIe começa num espaço após um mnemônico. Quando um operando é utilizado, sua finalidade é expandir um mnemônico em uma instrução completa. Alguns mnemônicos, como CLC, CLD e RTS, não requerem operandos; outros, como LDA, STA e ADC, sim. Abordaremos os operandos, com maiores detalhes, no Capítulo 6.

O Campo Comentários

Embora os comentários possam começar no campo rótulo de um programa em assembly, podem, também, à direita do campo Operandos, em campo próprio. Os comentários são utilizados, nos programas em assembly, de forma similar à utilização dos lembretes nos programas em BASIC: eles não afetam o programa, mas são usados para explicar procedimentos de programação e para inclusão de espaços em branco nas listagens, facilitando a leitura.

O PROGRAMA ADDNRS.SRC, LINHA POR LINHA

Já observamos o programa campo por campo; vamos agora observá-lo linha por linha. Se o programa foi digitado corretamente, e listado com o comando L, aparecerá na tela conforme ilustra o Programa 4.3

Programa 4.3
O PROGRAMA ADDNRS.SRC
(versão Apple ProDOS Assembler)

```
 1                    ORG     $1000
 2      ;
 3      ; ADDNRS.SRC
 4      ;
 5      ADDNRS        CLD
 6                    CLC
 7                    LDA     #2
 8                    ADC     #3
 9                    STA     $0300
10                    RTS
```

Linha 1: A Diretiva ORG A linha 1 é a *linha de origem* do programa. Quando uma linha ORG é usada num programa em assembly, ela diz onde o programa deverá ser armazenado na RAM, após a conversão em código-objeto. Nem todos os assemblers requerem uma diretiva ORG (ela é opcional, por exemplo, no Merlin Pro e no ORCA/M).

Mas o Apple ProDOS assembler requer que cada programa tenha uma diretiva de origem. Caso ela não exista, o assembler não gera nenhum código-fonte.

Nem sempre é fácil decidir onde o programa deve iniciar na memória, principalmente quando se é novato em programação em linguagem assembly. Seu computador possui vários blocos de memória que não podem ser utilizados para programas em assembly, por estarem reservados para outros usos, como conter o sistema operacional do computador, o sistema operacional de disco e o interpretador BASIC. Os próprios assemblers localizam-se em determinados blocos de memória; caso acidentalmente sobrescrevamos o programa do assembler, ele não mais poderá compilar o nosso programa.

Decidir onde armazenar um programa na memória é uma tarefa ardilosa com muitas variáveis a serem levadas em conta. Este tópico será objeto do Capítulo 11, dedicado somente ao gerenciamento de memória. Por enquanto, é suficiente lembrar que, se digitarmos os programas deste livro exatamente como apresentados, eles não irão se aventurar pelas áreas reservadas da RAM do computador.

Linhas de 2 a 4: Lembretes Os ponto-e-vírgulas que precedem o texto nas linhas de 2 a 4 mostram que essas linhas são constituídas somente por lembretes. A linha 3 dá o nome do programa. As linhas 2 e 4 servem somente para contornar a linha de título, destacando-a e facilitando a leitura.

É uma prática recomendável em linguagem assembly – assim como na maioria das linguagens de programação – a livre utilização de comentários e lembretes. Serão encontrados com freqüência nos programas deste livro.

Linha 5: Um Rótulo e um Op Code Como se pode ver, o campo rótulo nesta linha, foi usado para associar o rótulo ADDNRS ao programa ADDNRS.SRC. O programa funcionará perfeitamente sem rótulo, mas, como já dissemos anteriormente, um programa torna-se mais útil quando tem um rótulo, uma vez que pode ser acessado através dele e usado como parte de um programa maior. Desta forma, é interessante colocar rótulos em rotinas e sub-rotinas importantes. O rótulo não só torna a rotina mais fácil de localizar, como também é um lembrete do que a rotina faz (ou, até que seu programa tenha sido depurado, o que a rotina *deveria* fazer...).

Como já mencionamos, o 65C02/6502B pode realizar operações aritméticas com dois tipos de números: números binários comuns e decimais codificados como binários (BCD). Uma grande quantidade de informação a respeito dos números binários e BCD será fornecida no Capítulo 10. Por ora, é suficiente lembrar que a aritmética binária é a mais utilizada em programas 65C02/6502B, e que o mnemônico CLD desativa o indicador

(flag) decimal do microprocessador 65C02/6502B, de forma que os cálculos possam ser realizados utilizando números binários. (O comando para ativar o indicador decimal é SED.)

Não é necessário desativar o indicador decimal antes de cada operação aritmética em um programa, mas antes da primeira operação de adição ou subtração. Desta forma, não será preciso preocupar-se com o fato do indicador ter sido ou não ativado durante um programa prévio.

Linha 6: Um Op Code Antes de realizar uma operação de adição, limpa-se o indicador de transporte (carry flag) com CLC. O bit de transporte é afetado por uma série de operações e devemos trabalhar de forma segura. Demora apenas meio milionésimo de segundo e ocupa somente um byte de RAM a desativação do indicador de transporte. Comparado ao tempo e energia que a depuração pode consumir, é uma bagatela.

Linha 7: Op Code e Operando(LDA#2) Antes que uma operação de adição tenha lugar, o acumulador deve ser carregado com um dos números a serem somados. No programa ADDNRS, LDA#2 é o comando que carrega o acumulador. Neste comando, o sinal "#" à frente do número 2 indica que esse 2 é um número, não um endereço. Se a instrução fosse LDA2, o acumulador seria carregado com o conteúdo do endereço de memória 0002.

Linha 8: Op Code e Operando (ADC#3) Nesta linha, o comando ADC é usado para somar o número do acumulador com o número 3. O mnemônico ADC (add with carry) significa "adicione com transporte". Quando esta instrução é utilizada em um programa, ela soma o valor especificado no operando com o valor do acumulador, mais o valor do bit de transporte (carry bit) no registro de status do processador.

Neste caso, a adição do bit de transporte não tem efeito, uma vez que ele foi desativado antes da operação de adição. Não há instrução, na linguagem assembly 65C02/6502B, que signifique "adicione sem transporte". Se desejarmos adicionar um número sem transporte, é só desativar o indicador de transporte do registrador de status e realizar uma operação "adicione com transporte".

Linha 9: Op Code e Operando (STA $0300) Esta linha armazena o conteúdo do acumulador – neste caso, a soma de 2 e 3 – no endereço de memória $0300. O símbolo "#" aqui não é usado antes do operando ($0300), pois neste caso o operando é um endereço de memória, não um número.

Linha 10: Um Op Code (RTS) Se o mnemônico RTS é usado no final de uma sub-rotina, ele funciona como a instrução RETURN do BASIC: termina a sub-rotina e

retorna ao corpo principal do programa, na linha subseqüente àquela onde a instrução JSR apareceu. Quando, porém, JSR é usado no final do corpo principal de um programa, como no caso presente, a instrução tem função diferente: em lugar de passar o controle do programa a uma linha diferente, ela encerra o programa e retorna o controle do computador ao próximo programa de maior prioridade, que estava no controle antes do início deste – usualmente um sistema de operação de disco (DOS), um editor teclado/tela ou um monitor de linguagem de máquina.

IMPRIMINDO O PROGRAMA

Ao terminar a digitação de nossa listagem em código-fonte, poderemos rodá-la na impressora, em duas etapas: primeiramente, deveremos abrir um canal para a impressora, digitando o comando **PR#** seguido pelo número de dispositivo (device number) da impressora. Se o computador é um Apple IIc, o número de dispositivo da impressora será sempre 1 (a não ser que a impressora esteja conectada à saída para modem (modem port), caso em que o comando será PR# 2). Se for um Apple II com impressora, o número que segue a instrução PR# será o do slot, no qual está instalada a interface para impressora (na maioria dos Apple IIe, essa interface é instalada no slot 1).

Após abrir um canal para a impressora, deveremos ativá-la, digitando o comando PTRON, que é a abreviatura de *printer on* (impressora ligada). Depois, poderemos digitar **LIST** ou **PRINT** (ou as abreviaturas **L** ou **P**) para obter uma listagem impressa de nosso programa.

SALVANDO O PROGRAMA

Sempre que escrevemos um programa em linguagem assembly utilizando o *Apple ProDOS Assembler Tools*, o código-fonte introduzido vai sendo armazenado em um *buffer de edição* (edit buffer), que se estende do endereço de memória $0800 ao $9BFF, um bloco de RAM com 37887 bytes, suficiente para um extenso programa em código-fonte.

Infelizmente, este é quase exatamente o mesmo bloco de memória que o módulo compilador do ProDOS assembler tem de utilizar ao compilar um programa. Como dois programas não podem ocupar um mesmo espaço de memória ao mesmo tempo, qualquer código-fonte presente no buffer de edição do Apple ProDOS deve ser eliminado antes que o compilador ProDOS possa ser carregado em RAM para compilar o programa.

Uma vez que a eliminação de um programa da memória representa sua destruição, deve-se gravá-lo em disco assim que seja escrito. Felizmente, o assembler/editor do

Apple ProDOS torna difícil apagar acidentalmente uma listagem em código-fonte. Ele não permite que carreguemos o módulo compilador na memória (o que apagaria tudo o que estivesse armazenado no buffer de edição) até que digitemos a palavra **NEW**. Deveremos, então, gravar em disco o código em que estivemos trabalhando, antes de digitar **NEW** e apagar o conteúdo do buffer de edição.

Vamos, então, salvar o programa ADDNRS.SRC em disco e, depois, iremos compilá-lo.

USANDO COMANDOS DO ProDOS

Não é difícil salvar um programa criado com o *ProDOS Assembler Tools*, uma vez que o seu módulo editor aceita a maioria dos comandos mais utilizados do ProDOS (como PREFIX, CREATE, DELETE, LOCK, UNLOCK, CAT, LOAD e SAVE). Para chamar quaisquer comandos aceitos com o *Apple ProDOS Assembler Tools*, digitamos o comando desejado com o assembler/editor no modo de comando (edit). Se compreendemos os princípios do ambiente ProDOS, é fácil salvar um programa usando qualquer nome de arquivo válido. Para gravar o programa ADDNRS.SRC recém-escrito, por exemplo, estabeleçamos o prefixo ProDOS que desejamos usar e digitamos

SAVE ADDNRS.SRC

Quando o disk drive apagar o LED, verifica-se se o programa foi gravado com êxito.

Digitamos o comando ProDOS

CAT

Então, se o programa ADDNRS.SRC aparecer no diretório do disco, poderemos limpar o buffer de edição, digitando

NEW

A operação eliminará o conteúdo do buffer, e estaremos prontos para compilar o programa ADDNRS.SRC.

COMPILANDO UM PROGRAMA COM O Apple ProDOS Assembler

Uma vez desimpedido o buffer de texto com **NEW**, a compilação de uma listagem em código-fonte que foi gravada em disco é fácil. Digitemos uma linha como esta:

ASM ADDNRS.SRC – O compilador Apple ProDOS, então, realizará duas tarefas: lerá o código-fonte chamado ADDNRS.SCR do disco no qual foi gravado, gerando um arquivo código-objeto a partir do arquivo código-fonte lido, e armazenará esse código-objeto no disco. A menos que se especifique outra forma de execução, o compilador gravará o arquivo código-objeto no mesmo disco onde leu o arquivo código-fonte, e assinalará na listagem em código-objeto um nome default, que é o mesmo nome da listagem em código-fonte, mais o sufixo .0.

Se temos um programa chamado ADDNRS.SRC gravado em um disco, é muito fácil convertê-lo em código-objeto, com o Apple ProDOS assembler: carreguemos (*boot*) seu assembler/editor, certifiquemo-nos de que o buffer de texto esteja vazio, e digitemos a linha **ASM ADDNRS.SRC.** O compilador gerará um código-objeto executável a partir do arquivo em código-fonte, e chamará o novo arquivo em código-objeto de ADDNRS.SRC. .0.

Experimente agora este procedimento. A seguir, discutiremos aspectos mais refinados do uso do Apple ProDOS assembler. Finalmente, mostraremos como executar o código-objeto que o compilador gerou a partir do programa ADDNRS.SRC.

USANDO PARÂMETROS OPCIONAIS

Se desejarmos, podemos instruir o compilador para salvar o código-objeto em outro disco (ou, mais especificamente, em um dado diretório ou subdiretório ProDOS), e podemos também escolher um nome, em lugar do nome default, para o arquivo em código-objeto.

Suponhamos, por exemplo, que temos gravado o programa ADDNRS.SRC em um volume chamado SRCVOL, e desejamos gravar o programa, na versão código-objeto, em um volume diferente, chamado OBJVOL. Podemos realizar este procedimento chamando dois parâmetros opcionais:

```
ASM /SRCVOL/ADDNRS.SRC,/OBJVOL/ADDRNS.OBJ
```

O programa em código-fonte gravado no volume SRCVOL, sob o nome ADDNRS.SRC, será então compilado em um arquivo em código-objeto no volume OBJVOL, sob o nome ADDNRS.OBJ. Além disto, uma listagem do programa compilado será apresentada na tela do monitor. O Programa 4-4 mostra como a listagem compilada aparecerá:

Programa 4-4
ADDNRS.SRC
Uma Listagem Compilada

```
SOURCE   FILE #01 =>SRCVOL/ADDNRS.SRC
----- NEXT OBJECT FILE NAME IS OBJVOL/ADDNRS.OBJ
1000:          1000    1              ORG   $1000
1000:                  2 ;
1000:                  3 ; ADDNRS.SRC
1000:                  4 ;
1000:                  5 ADDNRS       CLD
1000:D8                6              CLC
1001:18                7              LDA   #2
1002:A9 02             8              ADC   #3
1004:69 03             9              STA   $0300
1006:8D 00 03         10              RTS
1009:60
#?1000 ADDNRS
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 14-DEC-83 15:21
** TOTAL LINES ASSEMBLED     10
** FREE SPACE PAGE COUNT    84
#
```

Examinando o Programa 4-4, notaremos que as linhas de 1 a 4 do programa ADDNRS não geram código-objeto, já que não contêm op codes ou operandos. A linha 1 contém um pseudo-op – ORG – que não realiza outra função a não ser informar o ProDOS assembler onde colocar o contador de programa ao iniciar a compilação. E as linhas de 2 a 4 contêm somente lembretes, que aparecem no programa ADDNRS em código-fonte e na listagem compilada, mas não geram código-objeto.

DIRIGINDO UMA LISTAGEM A UM DISK DRIVE OU IMPRESSORA

Quando o compilador/editor ProDOS compila um programa para Apple IIc ou IIe, uma listagem como a do Programa 4-4 é, normalmente, apresentada na tela. Entretanto, o ProDOS assembler possui o comando PR#, que pode ser usado para dirigir a listagem a outro dispositivo de saída, como uma impressora ou um disk drive. Quando uma listagem compilada é gravada em disco, ela é escrita na forma de um texto ASCII comum, não como um arquivo binário ou código-objeto executável.

Para armazenar uma listagem em disco, em lugar de apresentá-la na tela, basta digitar o comando PR# seguido do número do slot do disk drive, uma vírgula, e um nome apropriado. Se o computador é um Apple IIc, o comando PR# será sempre seguido do

número 6. Se for um Apple IIe, o número que segue o PR# será o número do slot para expansões no qual está instalada a interface para disk drive (usualmente o conector 6).

O procedimento todo não é tão complicado quanto pode parecer à primeira vista. Vamos supor que o cartão de controle de disk drive esteja instalado no conector 6. Vamos também supor que temos um programa em código-fonte chamado /SRCVOL/ADDNRS.SRC, e que desejamos compilá-lo em um programa em código-objeto chamado OBJVOL/ADDNRS.OBJ. Finalmente, vamos supor que desejamos salvar a listagem compilada do programa sob o nome /SRCVOL/ADDNRS.ASM. Podemos realizar tudo isso digitando duas linhas:

```
PR#6, /SRCVOL/ADDNRS.ASM
ASM/SRCVOL/ADDNRS.SRC, /OBJVOL/ADDNRS.OBJ
```

Simplesmente digitamos essas duas linhas no assembler e, se os volumes necessários estiverem nos disk drives corretos, o assembler deverá:

- Compilar o programa em código-fonte chamado /SRCVOL/ADDNRS.SRC.
- Gravar a listagem compilada sob o nome /SRCVOL/ADDNRS.ASM.
- Gravar o código-objeto gerado pelo compilador sob o nome /OBJVOL/ADDNRS.OBJ.

Caso ainda não o tenha feito, compilamos e salvamos um programa ADDNRS.OBJ. Antes, devemos realizar quaisquer ajustes de prefixo ProDOS de que precisamos e, então, digitamos a linha

```
ASM ADDNRS.SRC, ADDNRS.OBJ
```

SUPRIMINDO A GERAÇÃO DE CÓDIGO-OBJETO

Se desejarmos apresentar uma listagem compilada na tela, ou imprimi-la, sem gerar um arquivo código-objeto, digitamos uma vírgula e o símbolo "@" após a linha que contém o comando ASM.

```
ASM ADDNRS.SRC, @
```

O Apple ProDOS assembler, então, imprimirá a listagem compilada do programa ADDNRS.SRC, mas não gerará um arquivo código-objeto.

O ASSEMBLER MERLIN PRO

O restante deste capítulo será dedicado à discussão dos assemblers Merlin Pro e ORCA/M. Assim, se estiver trabalhando com um Apple ProDOS Assembler, você poderá pular o resto deste capítulo, e começar a ler o Capítulo 5. Se você possuir um ORCA/M, poderá ir diretamente à secção final deste capítulo, onde aprenderá como escrever e compilar um programa utilizando o sistema ORCA/M.

Existem muitas similaridades entre o Apple ProDOS Assembler e o Merlin Pro. Ambos utilizam números de linha relativos e possuem muitas funções de edição em comum. Os pseudo-ops usados pelos dois sistemas são quase idênticos.

Existem, também, muitas diferenças entre os sistemas. O Merlin Pro, diferentemente do Apple ProDOS Assembler, é comandado por menus. E o Merlin é totalmente compatível com os jogos de instruções dos microprocessadores de 16 bits 65802 e 65816, tanto quanto o é com os jogos de instruções do 6502B (Apple IIe) e do 65C02 (Apple IIC).

O Merlin apresenta também diversas características especiais – por exemplo, um excelente utilitário de encadeamento (linking) e vários comandos adicionais de edição.

O Merlin Pro também é acompanhado por uma extensa biblioteca de *macros* – rotinas pré-escritas em linguagem assembly que podem ser inseridas automaticamente nos programas escritos pelo usuário, economizando um bom tempo de programação (e digitação). Outro bônus do Merlin é um ótimo *descompilador* (disassembler), chamado The Sourceror.

Um disassembler é um utilitário para converter programas em linguagem de máquina de volta ao código-fonte. Para descobrir como um programa em linguagem de máquina funciona, usamos o disassembler para executar um pouco de "engenharia reversa". O Sourceror do Merlin é um dos melhores disassemblers existentes.

Além do Sourceror, o pacote Merlin Pro inclui um utilitário ainda mais especializado, o *Floating Point Sourceror* ou *Sourceror.FP*. Com ele, podemos obter uma listagem completa, de 150 páginas, do Apple-soft BASIC residente em seu Apple IIe ou IIc.

MÓDULOS DO MERLIN

Ao adquirir um Merlin Pro, recebemos dois discos. Ambos podem ser copiados e deveremos fazer cópias *backup* o mais cedo possível. Um dos discos é formatado usando o DOS

3.3 (o sistema operacional de disco que a Apple usou nos computadores da série II antes do desenvolvimento do ProDOS). O outro é um disco ProDOS e será o discutido nesta secção. Os programas nesse disco estão divididos em cinco módulos:

- *módulo executivo*
- *módulo editor*
- *módulo compilador* (assembler)
- *módulo de encadeamento* (linker)
- *módulo interpretador de comandos*

Quando o Merlin está no modo executivo, um menu-mestre é mostrado na tela, e quaisquer dos outros módulos podem ser selecionados desse menu. Para criar ou editar um programa em assembly, selecionamos o módulo editor. Para compilar um programa, usamos o módulo compilador. O módulo de encadeamento permite unir programas e pode-se chamar a maioria das funções de arquivo do ProDOS a partir do módulo interpretador de comandos.

O MENU DO MERLIN

Devemos carregar o Merlin a partir do disco; a primeira tela apresentar-se-á aproximadamente assim:

```
1
MERLIN-PRO 1.34
By Glen Bredon
a
(DF
C: Catalog
L: Load source
S: Save source
A: Append file
D: Disk command
E: Enter ED/ASM
O: Save object code
a: Set date
Q: Quit

%_
```

Todas as opções deste menu são explicadas com detalhes no manual de instruções que acompanha o pacote Merlin Pro Assembler. Por ora, notamos que o Merlin pode fazer pouca coisa no modo executivo, incluindo carregar e salvar código-fonte e código-objeto

e listar o conteúdo de um disco (usando-se o comando C do menu). Podemos ler e escrever arquivos-texto usando os comandos R e W. Podemos, também, formatar discos, apagar arquivos de discos e realizar várias outras funções de gerenciamento de disco, usando o comando D do menu.

Na parte inferior da tela, veremos um símbolo "%" seguido por um cursor piscante. O *prompt* "%" sempre indica que o Merlin se encontra no modo executivo. Quando no modo editor, o *prompt* muda para dois-pontos; no modo monitor, o *prompt* é um cifrão.

Após o *prompt* "%", teclamos

```
E
```

para entrar no modo Editor/compilador. Teclamos RETURN para ativar o módulo editor. Para indicar que se encontra no modo editor, o sistema apresentará o prompt ":" na tela. Teclemos

```
A
```

para "adicionar" (add). Veremos o número 1 aparecer na tela, que representa o primeiro número de linha a ser usado em um programa em código-fonte. O Merlin gera automaticamente os números de linha, começando do número 1 e incrementando unitariamente.

Se o sistema está funcionando corretamente deverá haver, após o 1, um espaço e um cursor piscante. Na posição do cursor, sem nenhum espaço adicional, teclamos um asterisco. A linha 1 de seu programa será assim:

```
1*
```

Teclemos RETURN e o Merlin avançará automaticamente para a linha 2. Após o 2, novamente sem nenhum espaço extra, digitemos

```
*ADDNRS.S
```

e pressionemos novamente a tecla RETURN. Ao aparecer a linha 3, teclemos outro asterisco. Vemos agora:

```
1*
2*ADDNRS.S
3*
```

Existe uma diferença óbvia entre as linhas acima e as três primeiras linhas do programa ADDNRS que foi criado com a utilização do Apple ProDOS Assembler. Na versão Apple ProDOS do programa, os comentários no campo Rótulo podiam começar

tanto por um asterisco quanto por um ponto-e-vírgula. Na versão Merlin, comentários no campo Rótulo sempre começam com asteriscos. Se tentarmos ali estabelecer um comentário com um ponto-e-vírgula, o editor irá aceitá-lo, mas será tabulado automaticamente para o campo Comentários (o quarto campo na tela).

Ao chegar à linha 4, teclamos RETURN e veremos o *prompt* ":" novamente. Teclamos, então, **A** (Adicionar) e continuamos a escrever o programa ou, digitar algum outro comando.

O assembler Merlin, assim como o ProDOS, listará o programa na tela, se teclarmos o comando **L** após o *prompt* de edição ":". O Merlin também elimina uma linha quando se tecla **D**. Para listar ou eliminar um grupo de linhas no Merlin, pode-se digitar o primeiro e o último números de linha após os comandos L ou D.

Ao utilizar o Merlin, deveremos separar os dois números que seguem o comando com uma vírgula, não com um traço. Eis o comando que, no Merlin, elimina as linhas 2 e 3:

```
D 2,3
```

Com esse comando poderemos restaurar as linhas eliminadas utilizando o comando A.

Outro comando que pode ser utilizado após o *prompt* ":" é I (Inserir). Tecle I, seguido do número de linha onde deseja que a nova linha seja inserida. Por exemplo, para inserir outro asterisco na linha 2 do programa ADDNRS.S, digite

```
I2
```

O Merlin responderá com o número 2.

Teclando então um asterisco, seguido de dois RETURNs, o *prompt* ":" aparecerá novamente e poderemos teclar **L** (List). O programa será listado e veremos que outra linha contendo um asterisco foi inserida em seu programa na linha 2.

O Merlin Pro, da mesma forma que o Apple ProDOS Assembler, usa numeração relativa de linhas e, portanto, renumera as linhas subseqüentes às linhas inseridas ou eliminadas. Deve-se inserir ou eliminar primeiramente as linhas de numeração mais alta, evitando assim a eliminação inadvertida das de numeração mais baixa.

Podemos agora eliminar o asterisco extra incluído no programa. Simplesmente teclamos RETURN, para obter um *prompt* dois-pontos, e digitamos **D2**. Após isso, teclamos **L** (List) e obteremos a listagem que mostra o programa restaurado à condição anterior:

```
1 *
2 *ADDRNRS.S
3 *
4
```

Além dos comandos A, I e D, existe também o comando **R** (Replace), que pode ser usado para substituir uma linha. Existem também comandos que podem ser usados para copiar linhas, encontrar e substituir seqüências (strings) e realizar várias outras funções, cujos detalhes de utilização se encontram no manual de instruções do Merlin Pro.

Vamos, agora, terminar a digitação do programa ADDNRS.S. O Programa 4-5 mostra o programa completo.

Programa 4-5
O PROGRAMA ADDNRS.S
(Como Digitado no Merlin Pro Assembler)

```
 1 *
 2 * ADDDNRS.S
 3 *
 4            ORG    $8000
 5 ADDNRS     CLD
 6            CLC
 7            LDA    #2
 8            ADC    #2
 9            STA    $02A7
10            RTS
11
```

O Merlin tabula entre as colunas automaticamente, dividindo o programa em campos de fácil leitura, e gera automaticamente um espaço após o número de linha, evitando que se precise teclar esse espaço; caso aconteça, pulará para a terceira coluna (na qual aparecem as abreviaturas de três letras).

A diretiva ORG é seguida do número $8000 no Merlin, em lugar do número $1000, que aparecia após a diretiva ORG na listagem do Apple ProDOS Assembler. A razão disto é que as configurações de memória dos dois assemblers são bem diferentes. Maiores detalhes sobre este assunto teremos no Capítulo 11.

LISTANDO SEU PROGRAMA

Ao atingir a linha 12 do programa ADDNRS.S, simplesmente teclamos RETURN. Poderemos, então, teclar **L** ou digitar **LIST** para ver o programa completo na tela.

OBSERVANDO MELHOR O PROGRAMA ADDNRS

O Programa 4-6 é uma visão detalhada do programa ADDNRS, escrito com o Merlin Pro. Neste exemplo, como no caso do Programa 4-2 deste capítulo, o programa é dividido em quatro campos ou colunas. Cada campo possui, no cabeçalho, uma descrição do tipo de informação nele contida.

Ao examinar a listagem, notamos que o Merlin não produz listagens que utilizam este tipo de espaçamento. Estas tabulações foram usadas para esclarecer melhor a organização de um programa em linguagem assembly.

Programa 4-6
O PROGRAMA ADDNRS.S
(Versão Merlin Pro)

```
NÚMERO           CÓDIGO
DE LINHA  RÓTULO OPERACIONAL  OPERANDO      LEMBRETES

 1 *
 2 * ADDDNRS.S
 3 *
 4            ORG          $8000
 5 ADDNRS     CLD
 6            CLC
 7            LDA           #2
 8            ADC           #2
 9            STA           $02A7
10            RTS
11
```

IMPRIMINDO O PROGRAMA

Ao terminar a digitação da listagem em código-fonte no editor do Merlin, podemos imprimi-la digitando o comando

PR#1

(Este comando funcionará no Apple IIc ou num Apple IIe cuja interface para impressora esteja instalada no slot de expansões 1. Se o cartão de impressora está instalado em outro conector, usa-se o número apropriado no comando.) Se desejarmos imprimir listagens com cabeçalho podemos também usar o comando PRTR. Para instruções mais detalhadas sobre a utilização destes comandos, há o manual de instruções do Merlin Pro.

COMPILANDO E GRAVANDO O PROGRAMA

Para compilar o programa ADDNRS.S usando o Merlim assembler, é só digitar o comando **ASM** após o *prompt* ":". O sistema perguntará se se deseja atualizar (update) o arquivo código-fonte (com a data de hoje, por exemplo). Se não, devemos teclar N (Não) e o Merlin rapidamente compilará nosso programa.

Antes de salvarmos o programa em disco, vamos comparar o código-objeto gerado pelo compilador com o código-fonte do qual foi derivado. No Programa 4-7, na coluna intitulada SOURCE CODE, veremos nossa listagem em código-fonte e, na próxima coluna, a versão em linguagem de máquina. Na coluna mais à direita, encontraremos o significado de cada uma das instruções.

Programa 4-7
O PROGRAMA ADDNRS.S
(Comparação Código-fonte/Código de máquina)

CÓDIGO-FONTE	CÓDIGO DE MÁQUINA	SIGNIFICADO
CLD	D8	Desativar o indicador de modo decimal do registro de status
CLC	18	Desativar o indicador de transporte do registro de status
LDA#2	A9 02	Carregar o acumulador com o número 2
ADC#3	69 03	Somar 3, com transporte
STA $0300	8D 00 03	Armazenar o resultado no endereço de memória $ 0300
RTS	60	Voltar da sub-rotina

Vamos, agora, salvar em disco tanto a listagem em código-fonte quanto a listagem em código-objeto. Primeiramente, tecle **Q** (Quit) após o *prompt* ":" para levar o sistema de volta ao modo executivo (menu). O menu principal reaparecerá. Poderemos, então, salvar o código-fonte selecionando S no menu e salvar o código-fonte selecionando O.

Ao teclar **S** ou **O**, o Merlin perguntará o nome que deverá ser dado ao programa. Não é necessário acrescentar sufixos para indicar que se trata de listagem em código-fonte ou programa em código-objeto: o sistema o fará automaticamente. Se estivermos gravando uma listagem em código-fonte, o assembler acrescentará um sufixo S ao nome; se estiver gravando uma listagem em código de máquina, não será acrescentado nenhum sufixo.

O ASSEMBLER ORCA/M

O assembler-editor ORCA/M é fabricado pela The Byte Works Inc., de Albuquerque, New Mexico. Se você tem um ORCA/M ou deseja aprender algo sobre ele, continue lendo este capítulo. Caso esteja usando um assembler Apple ou Merlin e esteja ansioso para rodar o programa ADDNRS, siga para o Capítulo 5.

O ORCA/M ("MACRO" soletrado do final para o início) é completamente diferente do Merlin Pro, do Apple ProDOS, e de quase todos os outros assemblers para microcomputadores. O ORCA/M, embora projetado para pequenos computadores, não é um "pequeno assembler".

A meta dos projetistas foi a de criar um assembler que trabalhasse como os utilizados nos grandes computadores (mainframes), e essa meta foi atingida com admirável sucesso. O ORCA/M é um dos melhores assemblers no mercado. E, a despeito de sua capacidade, é muito fácil de operar.

O sistema consiste de dois disquetes, ambos formatados em ProDOS. Como ambas as faces dos disquetes são utilizadas, o ORCA/M é, na realidade, um programa de quatro discos. Na face 2 do primeiro disco, existem diversos utilitários ORCA e ProDOS. Na face 1 do segundo disco, há uma gigantesca biblioteca de macros-rotinas prontas, em linguagem assembly, que podem ser inseridas automaticamente em programas escritos pelo usuário, economizando tempo de programação e digitação. No lado 2 do segundo disco, encontramos uma grande quantidade de sub-rotinas úteis, algumas das quais projetadas para trabalhar com as macros da face 1 do disco.

O ORCA/M é extraordinariamente bem projetado; aprendendo a utilizar o sistema, ele poderá realizar uma boa parte do trabalho de programação. Com o ORCA/M, podemos escrever programas longos em forma de pequenos módulos de fácil manuseio. Quando todos os módulos estiverem prontos, poderemos compor o programa através da utilização de um elegante sistema de encadeamento (linker). O ORCA/M trabalha não somente com o 65028 e o 65C02, mas com seus descendentes, os novos microprocessadores de 16 bits 65802 e 65816.

O ponto mais fraco do sistema ORCA/M é o manual de instruções. Mesmo com um bom conhecimento de linguagem assembly, não é fácil seguir o manual. A discussão a seguir destina-se a ajudá-lo a compreender o suficiente para iniciar a programação com o ORCA/M.

CARREGANDO O ORCA/M

Ambos os discos do pacote ORCA/M podem ser copiados. Faça as cópias e guarde os originais em lugar seguro. (É altamente recomendável equipar seu Apple com, pelo menos, dois disk drives, antes de tentar utilizar o programa ORCA/M; trata-se de um programa sofisticado, que efetua acessos alternados ao disco e à memória, durante a execução. Um par de disk drives – ou até mesmo um disco rígido (hard disks) – pode economizar muito tempo na utilização do ORCA/M.)

Coloque sua cópia do disco do programa ORCA/M no drive embutido de seu Apple IIc ou no drive 1 de seu Apple IIe. Inicialize, então, seu disco de programa, conforme as instruções do *System Inicialization Manual* do ORCA/M.

Coloque-o no drive 1 e coloque um disco vazio, mas formatado em ProDOS, no drive 2. Carregue, então (boot), seu disco de programa duplicata, como o faria, com qualquer disquete ProDOS. Aparecerá uma tela-título:

```
1
ORCA/HOST 4.0
Copyright (C) January 1985
By The Byte Works, Incorporated
@

      #_
```

O *prompt* "#" indica que o ORCA/M se encontra no chamado *modo processador de comandos*. Quando nesse modo, o programa aceita uma série de comandos, chamados *comandos de monitor*. Este nome pode causar confusão, uma vez que o monitor do ORCA/M não guarda nenhuma relação com o monitor de linguagem de máquina residente em seu Apple.

Descrições dos comandos de monitor que podem ser enviados ao processador de comandos do ORCA/M podem ser encontradas nas páginas 59 a 72 do manual de instruções daquele. Nesta secção, focalizaremos somente três desses comandos: PREFIX, EDIT e NEW.

PREFIX é um comando muito conveniente, já que pode ser usado para mudar o volume ProDOS no qual o ORCA/M acessa arquivos. Se você tem um sistema com dois drives, pode instruir o ORCA/M para acessar o drive 2 digitando

```
PREFIX .D2
```


assim que a tela-título aparece. Daí em diante, o assembler carregará e salvará arquivos código-fonte e código-objeto utilizando o drive 2.

OS COMANDOS NEW E EDIT

Após carregar ORCA/M e obter a apresentação da tela-título, digite o comando NEW após o *prompt* "#". O comando NEW indica ao sistema que você deseja entrar em modo editor, mas não tem ainda um arquivo a ser carregado. Se você desejar primeiramente carregar um arquivo na memória e então entrar no modo editor, digite **EDIT** após o *prompt* "#", seguido pelo nome do arquivo desejado:

```
#EDIT NOME COMPLETO DO ARQUIVO (PATHNAME)
```

Quando você digitar NEW (ou EDIT seguido pelo nome do arquivo), o disk drive será acionado, e logo apresentar-se-á uma tela vazia, exceto por um cursor piscante no canto superior direito e por uma linha indicadora de status na parte inferior da tela. Examine essa linha, apresentada em vídeo inverso, e verá uma série de circunflexos (^) em diferentes intervalos e uma linha de texto, que indicará em que linha e coluna se encontra o cursor, a quantidade de memória até então utilizada e o nome (se este existir) do arquivo em código-fonte apresentado na tela.

EDITOR DE TELA E NÚMEROS DE LINHA

Ao digitar um programa no ORCA/M, não é necessário preocupar-se com números de linha: eles não existem. Além disso, o editor do ORCA/M é um editor de tela, não um editor de linha, de forma que se podem usar as teclas-seta para movimentar o cursor pela tela, como o faria em um processador de texto.

O ORCA/M apresenta mais coisas em comum com um processador de texto: ele pode copiar blocos de texto de uma parte de um programa para outra, pode eliminá-los e movimentá-los. Para copiar, mover e eliminar blocos de texto, temos de usar seqüências com a tecla ESCAPE, descritas em detalhe no manual de instruções do ORCA/M.

O editor também aceita muitos comandos com a tecla CONTROL(^). Por exemplo, podemos mover o cursor para a parte superior (Top) da tela com um ^T, e para a parte inferior (Bottom) com um ^B. Podemos ir para a primeira linha de um arquivo com um ^F (First) e para a última, com um ^L (Last). Podemos inserir uma linha em qualquer lugar do programa, movendo o cursor para a posição desejada e digitando a seqüência ESCAPE-B, e eliminar uma linha com a seqüência ESCAPE-Y.

Outras funções control e escape que podem ser utilizadas com o editor ORCA/M são descritas no manual e no cartão de consulta rápida que acompanham o pacote ORCA/M. Estamos prontos, agora para algumas experiências com o ORCA/M. Digite a listagem do Programa 4-8; ele é chamado ADDNRS.SC, e é a versão ORCA/M do programa ADDNRS.

Programa 4-8
O PROGRAMA ADDNRS.SC
(Versão ORCA/M)

```
          KEEP  ADDNRS

ADDNRS    START

          CLD
          CLC
          LDA   #2
          ADC   #3
          STA   $0300
          RTS

          END
```

A versão ORCA/M do programa ADDNRS.SC apresenta algumas linhas que não aparecem nas versões Apple e Merlin. A primeira linha onde se lê "KEEP ADDNRS" é uma delas. Há uma outra, onde se lê "ADDNRS START" e ainda mais uma, onde vemos "END".

Note, também, que falta uma linha ORG, pois os programas escritos com o ORCA/M não necessitam de linhas de origem. Se se fornecer uma linha ORG ao ORCA/M, ele a utilizará: se não, o assembler designará automaticamente ao programa o endereço de início $2000, e este é, usualmente, um bom local para se iniciar um programa ORCA/M.

Voltemos às linhas extra do programa.

DIRETIVAS KEEP, START E END

KEEP é um pseudo-op freqüentemente usado em programas ORCA/M. Se uma diretiva KEEP precede um programa, o ORCA/M salvará o código-objeto produzido pela compilação do programa. A palavra usada como operando da diretiva KEEP será o nome no qual será gravado o código-objeto. Se não utilizarmos KEEP, o código-objeto não será gravado.

START E END são duas diretivas também freqüentemente encontradas nos programas ORCA/M. São usadas para delinear os pontos de início e fim de segmentos de código, com nome, que podem ser encadeados, mais tarde, para compor um longo programa. Uma vez que START tem a função de identificar um segmento de código, requer um título – neste caso, ADDNRS, o nome do programa. Não é necessário um título para END.

COMENTÁRIOS NOS PROGRAMAS COM O ORCA/M

A maneira pela qual o ORCA/M trata os comentários merece especial atenção. Num programa com o ORCA/M, uma linha de comentário pode ser precedida por um símbolo#, um ponto-e-vírgula ou um ponto de exclamação, mas linhas em branco são também tratadas como comentários. Uma linha em branco tem um propósito, é lógico – facilitar a leitura do programa, criando espaços entre os diversos segmentos. Espaços em branco podem ser bastante eficazes numa listagem, como se pode observar no Programa 4.8.

COMPILANDO O PROGRAMA ADDNRS.SC

Ao terminar a digitação do ADDNRS.SC, digitamos um control-q (a tecla control e a tecla Q, simultaneamente) e o ORCA/M responderá com esta tela:

```
====================================

        <R> Return to Editor
        <S> Save to the Same Name
        <N> Save to a New Name
        <L> Load Another File
        <E> Exit Without Updating

====================================

        Enter Selection: _
```

Este menu é destinado, basicamente, a permitir ao usuário carregar e gravar programas. Como o programa escrito ainda não tem um nome, a melhor maneira de gravá-lo será selecionar a opção N do menu ("Gravar com um Novo Nome"). Ao teclar N, a frase "Enter Selection:", na parte inferior da tela, mudará para "File Name:" (Nome do Arquivo:). Digitamos, então, **ADDNRS.SC**, e o código-fonte ADDNRS será salvo sob aquele nome.

Após salvar o programa ADDNRS.SC, colocamos o ORCA/M de volta em modo monitor, teclando **E** ("Exit without Updating"). O *prompt* "#" aparecerá novamente na tela. Digitando a palavra

```
ASSEMBLE
```

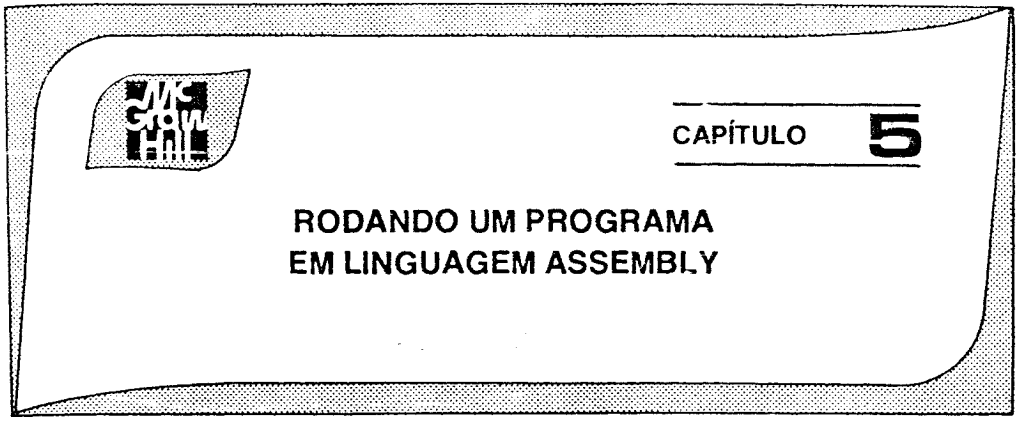
após o *prompt*, o programa ADDNRS.SC será então convertido em linguagem de máquina, e o código-objeto será gravado automaticamente no disco, sob o nome ADDNRS (a palavra que foi usada como operando da diretiva KEEP, na listagem).

IMPRIMINDO UMA LISTAGEM EM ASSEMBLY

Ao compilar um programa com o ORCA/M, podemos enviar a listagem em assembly para uma impressora, em lugar de enviá-la para a tela, com um comando como

```
ASSEMBLE > . PRINTER
```

No próximo capítulo, aprenderemos como rodar um programa compilado.



CAPÍTULO 5

RODANDO UM PROGRAMA EM LINGUAGEM ASSEMBLY

Tendo escrito e compilado um programa em linguagem assembly, o que obtemos é um programa em linguagem de máquina. Existem vários métodos que podem ser utilizados para executá-lo.

Pode-se rodar um programa usando o comando BRUN do ProDOS/BASIC, que existe somente com a finalidade de carregar e rodar programas em linguagem de máquina. Existe outro comando do DOS, BLOAD, destinado a carregar o código de máquina para execução posterior, quando e como se desejar.

Os comandos BRUN e BLOAD são muito versáteis, podendo ser utilizados como comandos diretos ou a partir de um programa em BASIC. Podemos, também, chamar o comando BLOAD a partir de muitos dos pacotes de software, incluindo o Apple ProDOS Assembler, o utilitário *Bugbyter* que o acompanha, e o Merlin Pro.

Carregado em RAM um programa em linguagem de máquina pode ser rodado usando o monitor de linguagem de máquina residente no computador. O monitor do Apple IIc/IIe tem um comando especial – o comando G – para rodar programas em linguagem de máquina.

Pode-se utilizar, também, o utilitário de depuração *Bugbyter*, similar ao monitor do Apple IIc/IIe, mas apresentando várias funções adicionais.

Para executar um programa em linguagem de máquina a partir de um programa em Applesoft BASIC, podemos utilizar o comando CALL, a função USR(X) ou comando G. (O pacote Apple ProDOS Assembler Tools contém um utilitário que efetua uma relocação do carregamento do programa, que pode auxiliá-lo a rodar programas em assembly a partir de um programa em BASIC. O Merlin Pro e o ORCA/M também produzem código

de máquina relocável, que pode ser chamado de programas em BASIC.)

Podemos, também, fazer com que um programa em linguagem de máquina seja carregado e rode automaticamente quando o disco é lido. Este procedimento pode ser realizado facilmente pela utilização do comando STARTUP do ProDOS/BASIC.

CARREGANDO UM PROGRAMA EM LINGUAGEM DE MÁQUINA

Antes de rodar um programa, é preciso carregá-lo na memória do computador, e existem várias maneiras de realizar isto: a partir de um ambiente ProDOS ou BASIC, por exemplo, pode-se carregar e rodar um programa em linguagem de máquina digitando

```
BRUN ADDNRS.OBJ
```

Se essa linha for introduzida como um comando direto BASIC/ProDOS, o computador procurará, em todos os volumes disponíveis do disco no drive default, por um programa em linguagem de máquina intitulado ADDNRS.OBJ. Caso o encontre, ele será carregado e rodado; se o nome não for encontrado, ele envia uma mensagem de erro.

O que foi descrito no parágrafo acima vale também para o que acontece quando o computador encontra, durante a execução de um programa em BASIC, uma linha como esta:

10 PRINT CHR$(4); "BRUN ADDNRS.OBJ"

(A sintaxe utilizada nesta linha é a "padrão" para a execução de comandos ProDOS a partir de programas em BASIC. Maiores detalhes sobre este assunto podem ser encontrados no manual *BASIC Programming With ProDOS*, da Apple, e em livros sobre ProDOS e programação em Applesoft.)

CARREGAR AGORA, RODAR DEPOIS

Se desejamos carregar um programa em linguagem de máquina, mas não desejamos que ele rode imediatamente, podemos utilizar o comando BLOAD do BASIC/ProDOS. Para utilizá-lo como um comando imediato, digita-se:

```
BLOAD ADDNRS.OBJ
```


Para utilizá-lo a partir de um programa em BASIC, usa-se o seguinte formato:

```
10 PRINT CHR$(4);"BLOAD ADDNRS.OBJ"
```

O programa será, então, carregado, mas não executado.

O comando BLOAD pode ser chamado não somente a partir do ProDOS ou de programas em BASIC, mas também do editor do Apple ProDOS Assembler e do utilitário de depuração *Bugbyter*. Maiores informações sobre estes dois métodos de utilização do comando BLOAD serão fornecidas mais adiante, neste capítulo.

EXECUTANDO UM PROGRAMA EM LINGUAGEM DE MÁQUINA

Estando carregado o programa em RAM, podemos executá-lo a partir do monitor do Apple IIC/IIe, do utilitário *Bugbyter* ou de um ambiente ProDOS ou BASIC. Comecemos pelo monitor:

O MONITOR DE LINGUAGEM DE MÁQUINAS DO APPLE IIc/IIe

Uma das formas mais simples para executar um programa em linguagem de máquina é a que utiliza o monitor residente no Apple. O monitor é uma utilíssima ferramenta de programação, existente desde o lançamento dos primeiros Apple. Com ele, podemos observar o que está na memória do computador, listando – e até mesmo alterando – o conteúdo das localizações de memória. O monitor pode traduzir o conteúdo das localizações de memória em linguagem assembly, e listá-la na tela. Pode, também, mover blocos de memória de uma localização a outra e comparar blocos de memória. E – o mais importante para nosso propósito atual – o monitor pode executar programas em linguagem de máquina.

O monitor pode ser chamado facilmente do Applesoft BASIC e também dos assemblers Apple ProDOS e Merlin. Podemos transferir o controle ao monitor, a partir desses assemblers, digitando MON e teclando RETURN.

Para chamar o monitor a partir do BASIC, digita-se CALL-151 ou CALL 65385 e tecla-se RETURN.

USANDO O MONITOR

Vamos, agora, carregar em RAM um programa em linguagem de máquina e executá-lo, utilizando o monitor.

Ligue seu computador e coloque o disco que contém o programa ADDNRS.OBJ no disk drive. Faça quaisquer ajustes nos prefixos ProDOS que sejam necessários e, então, digite **BLOAD ADDNRS.OBJ** e tecle RETURN. A seguir, digite o comando **CALL-151** e pressione RETURN novamente. Você deverá ver o *prompt* asterisco (*), que indica a presença do monitor.

Ao *prompt* "*", digite **4096** e tecle RETURN. Esse número é o equivalente decimal do número hexadecimal $1000, que é o endereço de início de seu programa ADDNRS.OBJ.

Quando o disk drive parar, você pode verificar se o programa ADDNRS.OBJ foi carregado com êxito digitando

```
*1000L
```

(O asterisco já estará lá.)

O L, no comando do monitor 1000L significa LIST. Quando digitamos esse comando e teclamos RETURN, o monitor *descompilará* até 20 linhas de instruções em linguagem de máquina, começando no endereço $1000, listando-as na tela. "Descompilar" significa traduzir instruções, em linguagem de máquina, de volta à linguagem assembly. Veremos a seguinte apresentação na tela:

"DESCOMPILAÇÃO", PELO MONITOR, DO PROGRAMA ADDNRS

```
COLUNA 1    COLUNA 2       COLUNA 3

1000-       D8             CLD
1001-       18             CLC
1002        A9 02          LDA #$02
1004-       69 03          STA #$03
1006-       8D 00 03       STA $0300
1009-       60             RTS
```

Abaixo desses endereços, veremos 14 linhas que mostram o conteúdo de endereços de memória que não fazem parte do programa ADDNRS. O comando L do monitor sempre cria uma apresentação de 20 linhas, e somente seis delas são necessárias para listar o programa ADDNRS. O conteúdo dos endereços extra poderá variar, dependendo dos programas que o computador estava rodando antes da chamada do monitor.

Os endereços extra podem conter apenas uma série de zeros, cada um deles seguido de um mnemônico BRK (uma instrução em linguagem assembly que equivale a um zero em linguagem de máquina).

Por outro lado, os endereços extra podem conter bits e partes de programas previamente carregados. Se os endereços agora ocupados pelo programa ADDNRS estavam ou não previamente ocupados por outro programa, não importa, pois como se trata de RAM endereçável pelo usuário, pode ser sobrescrita livremente. Como o programa ADDNRS termina com uma instrução RTS, o conteúdo dos endereços seguintes não tem importância.

Os números na Coluna 1 da ilustração são os endereços de memória hexadecimais nos quais o programa ADDNRS.OBJ está armazenado. Os números na Coluna 2 são instruções em linguagem de máquina (hexadecimal), armazenadas a partir dos endereços listados na Coluna 1. Como se pode observar, a Coluna 3 fornece uma listagem do programa ADDNRS.OBJ em linguagem assembly.

RODANDO UM PROGRAMA COM A UTILIZAÇÃO DO MONITOR

Após chamar uma listagem "descompilada" do programa e tê-la examinado, você constatou que o programa foi carregado com êxito na memória. Mas só saberá se o programa realmente funciona quando executá-lo.

É fácil executar um programa usando o monitor do Apple. Antes de fazê-lo, porém, vamos dar uma olhada no conteúdo de um outro registro de memória – o do endereço $0300. Este registro, como vimos, é o utilizado para armazenar a soma de 2 e 3 no programa ADDNRS. Vamos verificar esse registro antes de rodar o programa.

Digite

```
*0300
```

O monitor mostrará o conteúdo do registro $0300:

```
0300- 00
*
```

Essa resposta indica que o registro contém um 0. Caso não contenha, poderemos colocar um 0 no registro digitando o número 0300 seguido de um dois pontos e um 0.

```
*0300:0
```

O comando dois pontos pode ser usado para colocar qualquer valor de 8 bits em quaisquer dos registros da RAM. Faça algumas experiências, agora, usando este comando para mudar o conteúdo do registro de memória $0300, para diferentes valores, e verificando se funcionou. Após as experiências, certifique-se de haver deixado um 0 no registro $0300.

Agora podemos executar o programa ADDNRS utilizando o monitor. Após o *prompt* asterisco, digite o número 1000 (endereço de início do programa) e a letra G.

```
*1000G
```

Tecle RETURN e o programa deverá rodar.

Se tudo estiver funcionando corretamente, não acontecerá muita coisa quando o programa rodar; o que se nota é o aparecimento de outro *prompt* asterisco na tela. Mas algo deve, definitivamente ter acontecido. Para verificar isso, digite 0300 após o *prompt* asterisco. Você deverá ver

```
0300- 05
```

Essa linha indica que o monitor executou com êxito o programa ADDNRS.OBJ.

Como este não é, na realidade, um livro sobre o monitor do Apple IIc/IIe, nós apenas abordamos levemente suas características neste capítulo. O monitor pode também mudar, mover e comparar o conteúdo de pequenos e grandes blocos de memória, apresentar e mudar o conteúdo dos registros do 65C02/6502B e até mesmo realizar aritmética hexadecimal de 8 bits. Detalhes sobre estas e outras características do monitor do Apple IIc/IIe podem ser encontrados no *Manual de Consulta* (*Reference Manual*) do computador.

O PROGRAMA ADDNRS.SR2

Vamos observar um par de programas em linguagem de máquina, que não somente demonstram como os programas podem ser carregados e gravados, mas também mostram como um programa, carregado na memória, pode fornecer uma saída para a tela.

Vejamos o Programa 5.1, uma versão aperfeiçoada do programa ADDNRS.SRC. Ele foi escrito com o uso do Apple ProDOS Assembler, mas rodará também com o

Merlin Pro e (com pequenas modificações, explicadas nos Capítulos 4 e 5) pode funcionar com o ORCA/M. Como foi baseado no programa ADDNRS, recebeu o nome ADDNRS.SR2.

Mas, se você tem um Merlin Pro, que automaticamente acrescenta o sufixo ".S" a qualquer listagem em código-fonte que ele grava, poderá nomear o programa ADDNRS2.S.

Programa 5-1

ADDNRS.SR2

```
 1  ORG $1000
 2 *
 3 * ADDNRS.SR2
 4 *
 5 HOME EQU $FC58
 6 PRHEX EQU $FDE3
 7 *
 8 ADDNRS JSR HOME
 9  CLD
10  CLC
11  LDA #2
12  ADC #3
13  STA $0300
14  JSR PRHEX
15  RTS
```

O programa ADDNRS.SR2 faz tudo que seu predecessor fazia e algo mais. Primeiramente, ele limpa a tela, o que o ADDNRS.SRC não fazia. Depois, soma 2 e 3 e armazena o resultado no registro de memória $0300. Apresenta uma função de saída, que mostra o resultado na tela.

A maneira mais fácil de digitar o programa ADDNR.SR2 é carregar o programa ADDNRS.SRC original e usar as características de edição de seu assembler para atualizá-lo, gravando, então, esta nova versão sob o nome ADDNRS.SR2.

UMA TABELA DE SÍMBOLOS

Ao iniciar a expansão do programa ADDNRS.SRC para o novo ADDNRS.SR2, as primeiras adições a serem efetuadas serão as linhas 5 e 6. Juntas, elas compõem o que é conhecido, em linguagem assembly, como uma tabela de símbolos (Symbol table). Tabelas de símbolos são usadas, nos programas em assembly, para definir constantes e variáveis; a maioria das tabelas de símbolos inclui também rótulos de linha. A maioria dos programas em assembly usa tabelas de símbolos, encontradas em todos os programas deste livro, a

partir de agora.

No programa ADDNRS.SR2, a tabela de símbolos nas linhas 5 e 6 define somente dois valores, ambos sendo constantes. Na linha 5, uma constante chamada HOME é definida como o valor hexadecimal $FC58; na linha 6, a constante PRHEX é definida como o valor hexa $FDE3. As duas constantes são definidas com o auxílio de um pseudo-op, EQU, que significa "igual" (Equals). Em uma tabela de símbolos, não é necessário usar o símbolo "#" para indicar que o valor é um número, pois todos os valores ali presentes são assumidos como números.

O que significam os números da tabela de símbolos? Você deve se lembrar de um pequeno programa em assembly, chamado HI.TEST, apresentado no Capítulo 1. Esse programa fazia uso de uma conveniente sub-rotina em linguagem de máquina, residente no Apple IIc/IIe.

Geralmente chamada COUT nos programas em linguagem assembly, essa sub-rotina existe em ROM, tanto no Apple IIc como no IIe, e inicia no endereço de memória $FDED, podendo ser incorporada em qualquer programa em assembly com um simmples comando JSR.

Quando a sub-rotina COUT é chamada, ela espera que o código Apple ASCII de um caractere digitado seja armazenado no acumulador do 65C02/6502B. Se o valor de um caractere válido se encontra no acumulador, esse caractere será apresentado na tela.

COUT é somente uma dentre as inúmeras sub-rotinas em linguagem de máquina presentes nos computadores Apple IIc e Apple IIe. Todas estas sub-rotinas são descritas nos "reference manuals" dos computadores. Se você continuar a estudar linguagens assembly após terminar a leitura deste livro, deverá dedicar-se a conhecer muito bem todas estas sub-rotinas; a maioria delas realiza funções extremamente úteis, e todas podem ser chamadas com uma única instrução em assembly: JSR. Com essas rotinas preexistentes, economizamos muito trabalho, energia e tempo.

Antes de prosseguir, sugerimos que você compile o programa ADDNRS.SR2 sob o nome ADDNRS.082 e armazene em disco ambas as versões, código-fonte e código-objeto. Iremos, então, trabalhar com o programa durante o restante deste capítulo.

Ao executar a versão expandida do programa ADDNRS, você observará com clareza como as rotinas residentes HOME e PRHEX são usadas. A rotina HOME, que aparece na linha 8, faz o mesmo que a instrução HOME do Applesoft BASIC: limpa a tela e coloca o cursor no seu canto superior esquerdo.

As linhas de 9 a 13 do ADDNRS.SR2 realizam exatamente as mesmas funções

realizadas pelas linhas de 5 a 9 do programa ADDNRS.SRC original: somam 2 e 3 e armazenam o resultado em $0300.

UMA SUB-ROTINA DE APRESENTAÇÃO NA TELA

A linha 14 é também algo novo, acrescentado ao programa original. Nesta linha, a sub-rotina residente PRHEX é usada para apresentar a soma de 2 e 3 na tela. Se você consultar a lista de sub-rotinas residentes, no manual de consulta de seu computador, descobrirá que, na realidade, PRHEX apresenta na tela um dígito hexadecimal. Como, no caso, o dígito 5 é escrito da mesma forma nos sistemas decimal e hexadecimal, não importa se a saída da rotina PRHEX é, em verdade, um número hexadecimal.

Na linha 15, o programa ADDNRS.SR2 termina, da mesma forma que o fazia seu precesssor, com a tradicional instrução RTS.

O COMANDO BRUN

Uma vez compilado um programa, a maneira mais fácil de rodá-lo é através do comando BRUN do ProDOS/BASIC. Para rodar o programa ADNRS.0B2 usando o comando BRUN, armazene o programa em um disco *startup* (um disco em que estejam gravados o ProDOS e um arquivo BASIC.SYSTEM). Carregue, então, o disco e digite o comando

```
BRUN ADDNRS.OB2
```

Assim que o disco for carregado no computador, o número 5 apresentará no canto superior esquerdo da tela.

CRIANDO UM PROGRAMA STARTUP

Se seu programa ADDNRS.SR2 funcionou bem com o comando BRUN, você pode facilmente fazer com que ele carregue e rode automaticamente quando se liga o computador. Com o disco que contém o ADDNRS.SR2 ainda no disk drive, digite

RENAME ADDNRS.SR2, STARTUP

e tecle RETURN. Após a operação do drive se completar, desligue o computador e, após alguns segundos, ligue-o novamente. Seu programa ADDNRS.SR2, agora chamado

STARTUP, deverá carregar e rodar automaticamente; após o drive emitir os ruídos habituais de quando carrega um programa, você deverá ver o número 5 aparecer no canto superior esquerdo da tela.

CRIANDO UM PROGRAMA STARTUP MELHOR

Você, provavelmente, não encontrará nenhuma utilidade em um programa *startup* que apresenta a soma de 2 e 3 no canto superior esquerdo da tela; porém, os mesmos princípios aplicados para transformar o programa ADDNRS em um *startup* podem ser empregados para criar rotinas *startup* mais úteis. O Programa 5-2, por exemplo, é um programa em linguagem assembly que se carregará automaticamente do disco e apresentará uma breve mensagem na tela.

Programa 5-2
UM PROGRAMA "HELO" EM ASSEMBLY

```
 1 *
 2 HELLO.SRC
 3 *
 4 *ORG $1000
 5 *
 6 COUT EQU $FDED ;ROTINA DE APRESENTACAO NA TELA
 7 HOME EQU $FC58 ;ROTINA PARA LIMPAR A TELA
 8 *
 9 HELLO JSR HOME ;LIMPAR A TELA
10 LDA#$C8 ;CARREGAR A LETRA 'H'
11 JSR COUT;APRESENTA EM TELA
12 LDA#$C9 ;CARREGAR A LETRA 'I'
13 JSR COUT ;APRESENTA EM TELA
14 LDA#$A1 ;CARREGAR UM PONTO DE EXCLAMACAO
15 JSR COUT ;APRESENTA EM TELA
16 RTS ;FIM DO PROGRAMA
```

Note que o Programa 5-2 é uma versão ligeiramente melhorada do programa HITEST.SRC apresentado no Capítulo 1. Com o que já sabe sobre linguagem assembly, você não deverá ter dificuldade em entendê-lo. Ele utiliza rotinas em linguagem de máquina, residentes em seu Apple, para limpar a tela e apresentar a mensagem "HI!". E termina, como a maioria dos bons programas em assembly, com uma instrução RTS.

Digite o programa, compile (como HI.TEST.OBJ) e execute-o; ele limpará a tela e colocará o cursor na posição "home" e apresentará, então,

HI!

na tela.

A mensagem não é muito extensa, mas ainda não estamos prontos para apresentar mensagens longas na tela, uma vez que ainda não discutimos como são manuseadas seqüências de texto (text strings) na linguagem assembly. Embora curta, a mensagem "HI!" já faz algum sentido; pelo menos, é melhor do que o 5 apresentado pelo programa ADDNRS. Assim, talvez seja melhor colocar de volta o nome original, ADDNRS.OBJ, e transformar o HELLO.OBJ em um programa *startup*.

Para restaurar o nome original do ADDRNS.0B2, digite

```
RENAME STARTUP,ADDNRS.0B2
```

Depois você poderá digitar:

```
RENAME HELLO.OBJ,STARTUP
```

Feito isto, você verá um "HI!" em lugar do misterioso 5, cada vez que carregar o disco. Mais tarde, quando discutirmos seqüências de texto, veremos como apresentar mensagens longas na tela.

RODANDO UM PROGRAMA EM ASSEMBLY A PARTIR DO BASIC

Agora que já vimos como criar um disco *startup* em linguagem assembly, estamos prontos para abordar um novo assunto: misturar BASIC e linguagem assembly.

Já sabemos que um programa em assembly pode ser carregado e executado a partir de um programa em BASIC, usando-se uma linha como esta:

```
10 PRINT CHR$(4);"BRUN PATHNAME"
```

Este método, entretanto, não é comumente usado para essa finalidade, por não apresentar muita versatilidade. Quando se utiliza um comando BRUN num programa em BASIC, o programa de linguagem de máquina que ele chama é sempre rodado imediatamente após ter sido carregado.

Este processo nem sempre é desejável, pois entrega imediatamente o controle a qualquer programa em linguagem de máquina recém-carregado. A técnica mais usual é a de carregar o programa em linguagem de máquina durante a fase de inicialização de um programa em BASIC, e executá-lo mais tarde.

Felizmente, existe um comando do BASIC, o comando BLOAD, que permite

carregar um programa em linguagem de máquina na memória, sem executá-lo. Carregado em RAM, ele poderá ser executado a qualquer momento, com o auxílio de duas outras instruções do BASIC: o comando CALL e a função USR(x).

Para carregar um programa em linguagem de máquina com a função BLOAD, digite uma linha segundo este formato:

```
10 PRINT CHR$(4);"BLOAD PATHNAME"
```

Ao encontrar uma linha como esta em um programa em BASIC, seu computador procurará no disco que está usando (ou no volume que você especificar) por um programa em linguagem de máquina com o nome que foi requisitado. Encontrando tal programa, o computador irá carregá-lo em RAM, mas não o executará. Você poderá executá-lo quando o desejar, durante o transcurso de seu programa em BASIC, através do comando CALL ou da função USR(x).

USANDO O COMANDO CALL

É muito fácil utilizar o comando CALL do Applesoft BASIC; tudo o que se tem a fazer é carregar em RAM um programa em linguagem de máquina e então digitar a palavra CALL, seguida do endereço decimal do programa carregado. CALL pode ser usado tanto como um comando direto quanto um comando de um programa em BASIC. Quando chamado de um programa em BASIC, esta é a sintaxe utilizada:

```
20 CALL 4096
```

Se esta linha fosse incluída em um programa em BASIC, e existisse um programa em linguagem de máquina armazenado na memória, começando no endereço decimal 4096 (hexadecimal $1000), o controle seria transferido ao programa em linguagem de máquina.

Tentemos isto agora, usando o programa STARTUP (antigo HELLO.OBJ) armazenado no disco startup. Certifique-se de que esse disco esteja no disk drive e digite este programa em BASIC, de apenas duas linhas:

```
10 PRINT CHR$(4);"BLOAD STARTUP"
20 CALL 4096
```

Digite a palavra RUN, e tecle RETURN. Seu programa STARTUP – que inicia no endereço decimal 4096 (ou endereço hexa $1000) – deverá rodar, apresentando "HI!" na tela. E como ele termina com uma instrução RTS, deverá devolver o controle do computador ao BASIC.

Antes de passarmos à função USR(x), vale a pena mencionarmos uma característica especial da função CALL: em Applesoft BASIC, um comando CALL pode ser expresso tanto em números positivos quanto em negativos. Se expresso em um número negativo, o Applesoft BASIC somará, automaticamente, o número 65536, para obter o equivalente positivo do endereço. Esta característica inusitada do Applesoft BASIC vem a calhar, pois os endereços CALL negativos são geralmente menores e mais fáceis de memorizar do que os equivalentes positivos. Por exemplo, pode-se acessar o monitor do Apple IIc/IIe digitando CALL 65385 ou CALL-151. O segundo comando é de memorização muito mais fácil, sendo o mais comumente utilizado. Por outro lado, o equivalente negativo do número 4096 é –61440, de forma que é preferível utilizarmos 4096 mesmo...

A FUNÇÃO USR(X)

A função USR(X) é um pouco mais complicada que o comando CALL, porém, é muito mais poderosa. USR significa *user-supplied routine* (rotina fornecida pelo usuário) e o propósito da função USR é o de fornecer aos *experts* em programação BASIC um método para rodar rotinas de alta velocidade, em linguagem de máquina, a partir de programas em BASIC.

Quando souber programar em BASIC e em assembly, você poderá usar a função USR(x) para realizar computações matemáticas muito lentas programadas em BASIC – ou computações que o BASIC é incapaz de manipular. Suponha, por exemplo, que tenha acesso a uma série de rotinas, em linguagem assembly, para realizar complexas seqüências de animação de caracteres em gráficos de alta resolução. Você pode usar a função USR(X) para chamar e rodar tais seqüências – à velocidade de linguagem de máquina – de seus programas em BASIC!

Usando a função USR(x), você substitui o X que aparece entre parênteses por qualquer valor desejado. Então, ao chamar a função USR(X), três coisas acontecem.

Primeira: o valor assinalado ao X será armazenado automaticamente em uma série de endereços específica na memória.

Segunda: o programa em BASIC que está em execução será temporariamente interrompido e o controle do computador entregue a um programa em linguagem de máquina – geralmente escrito pelo usuário – que foi selecionado anteriormente.

Finalmente, quando a rotina trabalhar, poderá recuperar o valor a ela entregue, via a função USR(X), e realizar qualquer computação desejada usando aquele valor. O resultado da computação pode então ser armazenado na mesma série de registros de memó-

ria da qual veio o valor original, e o controle devolvido ao BASIC. Note que, uma vez que uma computação em linguagem de máquina tenha sido realizada na variável X da função USR(X), essa variável terá outro valor quando o controle for devolvido ao BASIC.

Para esclarecer este procedimento, iremos examinar dois programas ilustrativos – um deles escrito em BASIC, e o outro escrito em linguagem assembly – que foram interfaceados através da função USR(X). O Programa 5-3, chamado DECHEX.BA2, é o programa em BASIC e utiliza a função USR(X) para chamar um programa em linguagem de máquina: o Programa 5-4, chamado DECHEX, SR2.

Antes de iniciar a análise destes dois programas, digite e grave o programa DECHEX.BA2 e digite, compile e grave o programa DECHEX.SR2. Ao gravar a versão código-objeto deste último, é recomendável seguir as mesmas convenções adotadas até agora, designando o nome DECHEX.0B2.

Quando utilizados em conjunto, os Programas 5-3 e 5-4 converterão números decimais de 16 bits em números hexadecimais, apresentando-os na tela. Estes dois programas, juntos, são consideravelmente menores que o programa de conversão decimal/hexadecimal em BASIC, apresentado no Capítulo 2. Além disso, são capazes de manusear números positivos ou negativos, enquanto o programa DECHEX do Capítulo 2 só trabalhava com números positivos. Há ainda mais diferenças entre o programa do Capítulo 2 e os Programas 5-3 e 5-4. Antes, porém, de examinar essas diferenças, examinemos os dois programas:

Programa 5-3
USANDO A FUNCAO USR(X) PARA CHAMAR
UM PROGRAMA EM LINGUAGEM DE MAQUINA

```
 10 REM          *** DECHEX.BA2 ***
 20 REM
 30 POKE 10,76: REM COLOCAR INSTRUCAO 'JUMP' ($4C) NO REGISTRO DE MEMORIA 10 ($0A)
 40 REM COLOCAR AGORA O ENDERECO DO DECHEX.0B2 NOS REGISTROS 11 E 12 ($0B E $0C)
 50 HI = INT (4096 / 256):LO = 4096 – HI * 256: POKE 11,LO: POKE 12,HI
 60 PRINT CHR$ (4);"BLOAD DECHEX.0B2"
 70 TEXT : HOME : PRINT "CONVERSAO DECIMAL/HEXADECIMAL"
 80 PRINT : PRINT "DIGITE UM NUMERO DECIMAL"
 90 PRINT "ENTRE -32767 E 32767:": PRINT
100  INPUT NR1$: IF VAL (NR1$) <–3276 OR VAL (NR1$)>32767 THEN 80
110 X = VAL (NR$1)
120  PRINT "HEXA: ";
130 Y =       INT ( USR (X))
140  PRINT : GOTO 80
```


Programa 5-4
UM PROGRAMA EM CÓDIGO-FONTE CHAMADO PELA FUNÇÃO USR(X)

```
1   *
2   * DECHEX.SR2
3   *
4   * ORG $1000
5   *
6   PRNTAX EQU $F941 ;ROTINA PARA MOSTRAR A E X EM HEXA NA TELA
7   FLPINT EQU $E10C ;ROTINA DE CONVERSAO FLPT/INT
8   *
9   DECHEX JSR FLPINT ;CONVERTER VALOR EM PONTO FLUTUANTE PARA INTEIRO NO
    ACUMULADOR
10  LDA $A1 ;BYTE INFERIOR DO RESULTADO
11  TAX;PRNTAX TOMA O BYTE INFERIOR DO REGISTRO X
12  LDA $A0 ;BYTE SUPERIOR DO RESULTADO
13  JSR PRNTAX; PRNTAX TOMA O BYTE SUPERIOR DO ACUMULADOR
14  RTS
```

x

O Programa DECHEX.BA2, Linha por Linha Vamos examinar o Programa 5-4, DECHEX.BA2, começando pelas linhas de 30 a 50:

```
30 POKE 10,76: REM COLOCAR INSTRUCAO 'JUMP' ($4C) NO REGISTRO DE MEMORIA 10($0A)
40 REM COLOCAR AGORA O ENDERECO DO DECHEX.0B2 NOS REGISTROS 11 E 12 ($0B E $0C)
50 HI = INT (4096 / 256):LO = 4096 – HI * 256: POKE 11,LO: POKE 12,HI
```

Estas três linhas realizam uma operação que é sempre necessária antes de se chamar a função USR(X). Quando a função USR(X) é usada, ela primeiramente verifica os registros de memória 10, 11 e 12 ($0A a $0C em notação hexadecimal); se encontrar um jogo de instruções em linguagem de máquina nestes três registros, essas instruções serão executadas. Se não forem encontradas instruções em linguagem de máquina em $0A a $0C, será emitida uma mensagem de erro, ou poderão ser produzidos resultados imprevisíveis, potencialmente desastrosos. Assim, antes de se chamar a função USR(X), é necessário armazenar três instruções em linguagem de máquina naqueles registros.

Na linha 30 do programa DECHEX.BA2, o op code $4C (76 em notação decimal) é armazenado no registro $0A. A instrução $4C é o equivalente, em linguagem de máquina, do mnemônico JMP em linguagem assembly. Este, como você deve se lembrar, é similar à instrução GOTO do BASIC. Quando JMP é utilizado em um programa em assembly, ele é sempre seguido por um endereço de memória ou por um título que equivale ao endereço. Ao encontrar um JMP, o programa pula para o endereço ali especificado.

Nas linhas 40 e 50 do programa, um valor de 16 bits é armazenado nas localizações de memória $0B e $0C (11 e 12 em decimal). Como se pode notar, o número arma-

zenado nesses dois registros é $1000 (ou 4096 em decimal). Observando o Programa 5-4 veremos que o valor $1000 que segue a instrução JMP é o endereço de início do programa em assembly DECHEX.SR2.

Portanto, o que as linhas 30 a 50 do DECHEX.BA2 fazem é ajeitar as coisas para que, quando a função USR(X) for mais tarde chamada no programa DECHEX.BA2, transfira o controle ao programa DECHEX.SR2.

Vejamos, agora, as linhas de 60 a 130:

```
 60 PRINT: CHR$ (4);"BLOAD DECHEX.0B2"
 70 TEXT : HOME : PRINT "CONVERSAO DECIMAL/HEXADECIMAL"
 80 PRINT : PRINT "DIGITE UM NUMERO DECIMAL"
 90 PRINT : ENTRE -32767 E 32767:":PRINT
100  INPUT NR1$: IF VAL (NR1$)<-32767 OR VAL (NR1$)>32767 THEN 80
110 X = VAL (NR1$)
120  PRINT "HEX:";
```

A linha 60 carrega um programa chamado DECHEX.0B2 (o nome do arquivo da versão código-objeto do programa DECHEX.SR2).

Nas linhas de 70 a 90, um título e duas linhas com instruções para o usuário são mostrados na tela. As instruções solicitam a digitação de um número e estabelecem que esse número deverá estar compreendido entre –32767 e 32767. O programa DECHEX.BA2, portanto, pode manusear também números negativos, o que não acontecia com o DECHEX do Capítulo 2.

Mas, existe um preço a ser pago por esta capacidade extra: como você deve-se lembrar, o programa DECHEX podia converter em hexadecimal qualquer número positivo, até 65535. Já o programa DECHEX.BA2 manuseia somente metade desse valor, ou seja, 32767. Como ele manuseia também números negativos até –32767, o número *total* de valores que ambos os programas podem traduzir para hexadecimal é o mesmo. (Aprenderemos mais a respeito de números negativos no Capítulo 10, que versa sobre a matemática da linguagem assembly.)

Na linha 100 do DECHEX.BA2, um comando INPUT é usado para aceitar o número digitado como uma seqüência (string). Se esse número for menor que –32767 ou maior que 32767, o programa se recusa a aceitá-lo e pede uma nova digitação. Uma vez introduzido um número válido, é convertido em um valor numérico, e então definido como o valor da variável X na função USR(X). Na linha 120, a mensagem "HEXA:" é apresentada na tela.

Usando a Função USR(X) Quando a função USR(X) é finalmente chamada, na linha 130, ela primeiramente deposita o valor de X em uma série especial de registros de memória, chamada *acumulador de ponto flutuante* (floating-point accumulator), que é um bloco de memória utilizado para armazenar números durante operações que envolvem um procedimento denominado *aritmética de ponto flutuante*. No Apple IIc e no Apple IIe, o acumulador de ponto flutuante situa-se nos registros de memória $9D a $A3.

No Capítulo 10, que trata especificamente da matemática 6502B/65C02, a aritmética de ponto flutuante será discutida em detalhe. Vamos, agora, utilizar mais duas sub-rotinas em linguagem de máquina, residentes em todo Apple IIc e IIe. Uma destas sub-rotinas, aqui chamada FLPINT, destina-se a converter números em ponto flutuante para números binários de 16 bits. A outra, aqui denominada INTFLP, trabalha em sentido inverso: converte binários de 16 bits em ponto flutuante. Eis uma breve explanação destas duas sub-rotinas:

A sub-rotina FLPINT, que inicia no endereço de memória $E10C, tomará o valor do acumulador de ponto flutuante e o converterá em um número em ponto flutuante, depositando o byte de ordem superior desse número no registro de memória $A0, e o byte de ordem inferior no registro $A1. Para utilizar esta sub-rotina, usamos a instrução em assembly JSR para pular ao endereço de início da sub-rotina, $E10C.

(*Nota:* À página 174, edição 1982 do *Applesoft BASIC Programmer's Reference Manual* do Apple IIe, o endereço da rotina FLPINT, fornecido como sendo $E01C, está incorreto. O endereço correto é o fornecido acima, $E10C.)

Para usar a rotina INTFLP, deveremos colocar um inteiro binário de dois bytes no acumulador e no registro Y do microprocessador 6502B/65C02. O byte inferior daquele número deverá ser colocado no acumulador, e o byte superior, no registro Y. Façamos, então, um jump, usando a instrução JSR, para a sub-rotina INTFLP, no endereço $E2F2. Após a execução da sub-rotina, o equivalente em ponto flutuante do número que foi armazenado nos registros A e Y estará presente no acumulador de ponto flutuante do computador.

Analisando o Programa DECHEX.SR2 Estamos, agora, prontos para examinar a linha 130 do DECHEX.SR2:

```
130 Y =  INT ( USER (X))
```

Esta linha chama a função USR(X); como vimos, essa função deposita o valor de X no acumulador de ponto flutuante. Depois transfere o controle a qualquer programa em linguagem de máquina que inicie no endereço que foi carregado nas localizações de memória $0B e $0C.

Como o endereço do programa DECHEX.SR2 está agora ali (pois foi armazenado quando da execução da linha 50 do programa DECHEX.BA2), o programa em linguagem de máquina para o qual a função USR(X) pulará é o programa DECHEX.SR2.

Vamos, então, observar, linha por linha, o programa DECHEX.SR2, começando pelas linhas 6 e 7:

```
6 PRNTAX EQU $F941 ;ROTINA PARA APRESENTAR A E X, EM HEXA, NA TELA
7 FLPINT EQU SE10C, ROTINA DE CONVERSAO FLPT/INT
```

Temos aqui outra tabela de símbolos, que designa rótulos aos endereços de início de rotinas importantes. Nessa tabela de símbolos de somente duas linhas, apenas dois endereços são listados. Na linha 6, o endereço de uma rotina chamada PRNTAX é definido como $F941 e, na linha 7, temos o endereço da rotina FLPINT, a qual converte números em ponto flutuante para números hexadecimais. Como já discutimos esta última rotina, vamos voltar a atenção para a rotina PRNTAX da tabela de símbolos do programa DECHEX.SR2.

A rotina PRNTAX, como a FLPINT, é residente nos Apple IIc e IIe e é uma rotina muito importante num programa como o DECHEX.SR2. Ela apresenta, automaticamente, números hexadecimais na tela. Para usar PRNTAX, devemos colocar um número hexadecimal de 16 bits nos registros A e X do microprocessador, com o byte de ordem superior no acumulador e o byte de ordem inferior no registro X. Quando, então, for executado um JSR ao endereço $F941, o número armazenado nos registros A e X será apresentado na tela.

Rodando o Programa DECHEX.SR2 Antes de continuarmos a discutir o restante do programa DECHEX.SR2, é importante recordarmos o que aconteceu até agora: um número foi digitado e armazenado no acumulador de ponto flutuante. Então, com o auxílio da função USR(X), o controle foi transferido para o programa DECHEX.SR2.

```
 9  DECHEX JSR FLPINT ;CONVERTER VALOR EM PONTO FLUTUANTE PARA INTEIRO NO ACUMULADOR
10  LDA $A1 ;BYTE INFERIOR DO RESULTADO
11  TAX ;PRNTAX TOMA O BYTE INFERIOR DO REGISTRO X
12  LDA $A0 ;BYTE SUPERIOR DO RESULTADO
13  JSR PRNTAX ;PRNTAX TOMA O BYTE SUPERIOR DO ACUMULADOR
14  RTS
```

Prossigamos, então: a sub-rotina FLPINT é chamada na linha 9 do programa DECHEX, e o valor no acumulador de ponto flutuante – isto é, o valor que foi digitado – é convertido em um inteiro de dois bytes. A rotina FLPINT armazena o byte inferior

desse inteiro em $A1 e o byte superior em $A0, encerra e devolve o controle ao programa DECHEX.SR2.

Na linha 10 do DECHEX.SR2, o acumulador é carregado com o valor de $A1 (o byte inferior do inteiro); e, na linha 11, há um mnemônico da linguagem assembly que ainda não havíamos encontrado: TAX, que significa "transfira o valor do acumulador para o registro X".

Na linha 11, o mnemônico TAX faz exatamente isso: move o valor que foi recém-carregado no acumulador para o registro X. Após a realização desta operação, portanto, o byte inferior do valor retornado pela rotina FLPINT está presente no registro X.

Restam somente mais três linhas no programa DECHEX.SR2. Na linha 12 o acumulador é carregado com o valor que se encontra no registro de memória $A0 (isto é, o byte superior do valor retornado por FLPINT). Na linha 13, há um salto para a rotina PRNTAX, que combina o valor do acumulador e o valor do registro X, apresentando-os na tela como um número hexadecimal de quatro dígitos.

A FUNÇÃO USR(X) COMO FERRAMENTA DE PROGRAMAÇÃO

A função USR(X) é uma das funções mais complexas, das disponíveis em BASIC, mas também uma das mais poderosas. Sabendo como utilizá-la, podemos compor um extenso repertório de funções em linguagem de máquina e usá-las à vontade em programas em BASIC. É muito mais fácil programar em BASIC do que em assembly.

Infelizmente, porém, o BASIC é tão lento, que simplesmente impossibilita o uso de vários tipos de rotinas, especialmente as que envolvem gráficos de alta resolução. A função USR(X), entretanto, nos permite movimentar entre BASIC e assembly; se soubermos utilizar a função USR(X), poderemos tirar proveito das características vantajosas de ambas as linguagens, utilizando BASIC para os tipos de rotinas para os quais apresenta melhor adequação e, sempre que desejar, poderemos usar a função USR(X) para chamar rotinas em assembly, de alto desempenho e velocidade.

CAPÍTULO 6

INSTRUÇÕES DO 6502B/65C02

Até agora, discutimos, principalmente, a sintaxe e a gramática da linguagem assembly do 6502B/65C02, e abordaremos estes tópicos com maiores detalhes em capítulos subseqüentes. Este capítulo, porém, é importante, pois trata do *vocabulário* da linguagem assembly do Apple IIc/IIe – o jogo de instruções (instruction set) do 6502B/65C02.

Este capítulo não foi elaborado para ser lido uma única vez; é um capítulo de consulta, de forma que você, provavelmente, irá apenas folheá-lo, para ter uma idéia do material nele contido, e irá consultá-lo quando necessário.

A maior parte deste capítulo é composta de uma listagem, em ordem alfabética, do jogo de instruções do 6502B/65C02. Essa listagem inclui todas as 56 instruções em linguagem assembly que são utilizadas com o microprocessador 6502B, e 8 instruções adicionais, usadas com o 65C02. Estas 8 instruções, específicas do 65C02, estão marcadas com asteriscos.

Os modos de endereçamento usados no 6502B e no 65C02 também estão listados, e os modos aplicáveis somente ao 65C02 têm uma identificação indicativa desse fato.

Instruções para o 65802 e o 65816 – os novos membros da família 6502, mas de 16 bits – não estão incluídas na listagem deste capítulo, mas podem ser encontradas nos Apêndices deste livro.

Para cada mnemônico da lista, há uma breve explanação sobre a sua utilização nos programas em assembly para Apple IIc e IIe, e uma indicação dos registros e indicadores (flags) afetados pelo mnemônico.

Ao final do capítulo, após a listagem do jogo de instruções, encontra-se um bônus especial: um programa em BASIC, chamado "Simulador de Bytes", que ajuda na compreensão de como é processado o jogo de instruções pelo microprocessador. Como o programa é escrito em BASIC, poderá ser digitado e rodado sem o uso de assembler. Ao rodá-lo, vemos porque ele tem esse nome.

ABREVIATURAS USADAS NESTE CAPÍTULO

A seção seguinte contém a lista completa do jogo de instruções do 6502B/65C02 – todos os mnemônicos utilizados na programação em assembly dos Apple IIc e IIe. A lista, é claro, não inclui pseudo-operações (também chamadas pseudo-ops ou diretivas), que variam de assembler para assembler.

Ela inclui os modos de endereçamento, que serão objeto de estudo no Capítulo 7. Para obter a lista de pseudo-ops usados por seu assembler, consulte o seu manual de instruções.

As Tabelas 6-1 a 6-3 definem as abreviaturas utilizadas no jogo de instruções.

Tabela 6-1 Indicadores do Registro de Status (P) do Processador

N	Negative (sign) flag	Indicador de sinal negativo
V	Overflow flag	Indicador de overflow
B	Break flag	Indicador de "break"
D	Decimal flag	Indicador de decimal
I	Interrupt flag	Indicador de interrupção
Z	Zero flag	Indicador de zero
C	Carry flag	Indicador de transporte

Tabela 6-2 Registro de Memória do 6502B/65C02

A	Acumulador
X	Registro X
Y	Registro Y
M	Registro Memória

Tabela 6-3 Modos de Endereçamento do 6502B/65C02

A	Absolute addressing	Endereçamento absoluto
AC	Accumulator addressing	Endereçamento do acumulador
AI	Absolute indexed indirect addressing (JMP instruction, 65C02 only)	Endereçamento indireto, absoluto indexado (Instrução JMP, somente 65C02)
Z	Zero-page addressing (65C02 only)	Endereçamento da página zero (Somente 65C02)
IMM	Immediate addressing	Endereçamento imediato
IND	Absolute indirect addressing	Endereçamento indireto absoluto
IMP	Implied addressing	Endereçamento implícito
AX	Absolute, X (X-indexed) addressing	Endereçamento absoluto, X (X-indexado)
AY	Absolute, Y (Y-indexed) addressing	Endereçamento absoluto, (Y - indexado)
IX	Indexed indirect (Indirect, X) addressing	Endereçamento indireto indexado (Indireto, X)
IY	Indexed indirect (Indirect, Y) addressing	Endereçamento indireto indexado (Indireto, Y)
R	Relative addressing	Endereçamento relativo
ZPG	Zero-page indirect addressing	Endereçamento indireto da página zero
ZX	Zero-page X-indixed (Zero, page, X) addressing	Endereçamento X-indexado da página zero (Página zero, X)
ZY	Zero-page Y- indexed (Zero-page, Y) addressing	Endereçamento Y-indexado da página zero (Página zero, Y)

O JOGO DE INSTRUÇÕES

ADC (Add with carry) Soma o conteúdo do acumulador ao conteúdo de uma localização específica de memória ou valor. Se o indicador de transporte do registro P está ativado, um transporte é também adicionado.O resultado da operação de adição é então armazenado no acumulador.

Indicadores afetados: N, V, Z, C
Registros afetados: A
Modos de endereçamento: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

AND ("E" lógico) Realiza uma operação lógica E com o conteúdo do acumulador e o conteúdo de uma localização específica de memória ou valor imediato. O resultado da operação é armazenado no acumulador.

Indicadores afetados: N, Z

Registros afetados: A
Modos de endereçamento: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

ASL (Arithmetic shift left) Mudança aritmética de casas para a esquerda: Move cada bit do acumulador ou de uma localização de memória especificada, uma posição para a esquerda. Um 0 é depositado na posição do bit 0, e o bit 7 é forçado para o bit de transporte (carry bit) do registro P. O resultado da operação é deixado no acumulador ou no registro de memória afetado.

Indicadores afetados: N, Z, C
Registros afetados: A, M
Modos de endereçamento: AC, A, Z, AX, ZX

BCC (Branch if carry clear) Desvio se o transporte está desativado: Executa um desvio se o indicador de transporte (carry flag) do registro P estiver desativado. Não resulta em nenhuma operação se o indicador de transporte estiver ativado.

O destino do desvio é calculado pela adição de um deslocamento com sinal, variando entre −128 e +127, ao endereço da primeira instrução subseqüente à instrução BCC. Este cálculo resulta num deslocamento efetivo de +129 bytes a −126 bytes da instrução de desvio (branch). Uma tentativa de utilização da instrução BCC para um desvio mais longo resultará num erro do tipo *out of range* (fora do alcance).

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: R

BCS (Branch if carry set) Desvio se o transporte está ativado. Executa um desvio, se o indicador de transporte (carry flag) do registro P estiver ativado. Não resulta em nenhuma operação se o indicador de transporte estiver desativado. A destinação do desvio é calculada pela adição de um deslocamento com sinal, variando entre −128 e +127, ao endereço da primeira instrução subseqüente à instrução BCC.

Isto resulta num deslocamento efetivo de +129 bytes a −126 bytes da instrução de desvio (branch). Uma tentativa de utilização da instrução BCS para um desvio maior resultará em um erro do tipo *out of range* (fora do alcance).

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: R

BEQ (Branch if equal) Desvio se igual: Executa um desvio se o indicador de zero (zero flag) do registro P estiver ativado. Não resulta em nenhuma operação se o indicador de zero estiver desativado. Pode ser usado para causar um desvio, se o resultado de um cálculo é 0 ou se dois números são iguais.

O destino do desvio é calculado pela adição de um deslocamento com sinal, variando de –128 a +127, ao endereço da primeira instrução subseqüente à instrução BEQ. Este cálculo resulta em um deslocamento efetivo de + 129 a – 126 bytes da instrução de desvio (branch). Uma tentativa de utilização da instrução BEQ para um desvio maior resultará num erro do tipo *out of range* (fora do alcance).

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: R

BIT (Comparar os bits do acumulador com os bits de um registro de memória especificado) Realiza uma operação binária lógica E com o conteúdo do acumulador e o conteúdo de um endereço de memória especificado. O conteúdo do acumulador não é afetado, mas três indicadores (flags) no registro P são afetados.

Se todos os bits presentes no acumulador coincidirem com todos os bits do valor que está sendo testado, o indicador Z é desativado. Se nenhum dos bits presentes no acumulador coincidir com os bits do valor sob teste, o indicador Z é ativado.

Assim, utilizando uma instrução BIT seguida de uma instrução BNE ou uma instrução BEQ, podemos determinar se há alguma coincidência entre os bits do acumulador e os do valor sob teste. Se existe uma coincidência, um teste BIT/BNE terá sucesso; se não há coincidência, um teste BIT/BEQ terá sucesso.

A instrução BIT produz ainda outro efeito: os bits 6 e 7 do valor testado são transferidos diretamente aos bits V e N do registro de status. Esta característica provê uma forma conveniente para a verificação dos bits 6 e 7 de qualquer valor desejado, em uma única operação; a instrução BIT é, por essa razão, freqüentemente utilizada em operações aritméticas que envolvem números com sinal.

Indicadores afetados: N, V, Z
Registros afetados: —
Modos de endereçamento: A, Z, IMM*, AX*, ZX*

BMI (Branch on minus) Desvio, quando sinal de menos: Executa um desvio se o indicador N do registro P estiver ativado. Não resulta em nenhuma operação se o indicador N estiver desativado. A destinação do desvio (branch) é calculada pela adição de um deslocamento com sinal, variando entre –128 e +127, ao endereço da primeira instrução subseqüente à instrução BMI.

Isto resulta em um deslocamento efetivo de +129 a –126 bytes a partir da instrução de desvio. Tentativas de utilização da instrução BMI para um desvio maior resultarão em um erro do tipo *out of range* (fora do alcance).

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: R

BNE (Branch if not equal) Desvio, se não for igual: Executa um desvio se o indicador 0 (0 flag) do registro P estiver desativado (isto é, se o resultado de uma operação for diferente de 0). Não resulta em nenhuma operação se o indicador de 0 estiver ativado. Pode ser usado para causar um desvio se o resultado de um cálculo for diferente de zero, ou se dois números não forem iguais. O destino do desvio é calculado pela adição de um deslocamento com sinal, variando de –128 a +127, ao endereço da primeira instrução subseqüente à instrução BNE.

Este cálculo resulta num deslocamento efetivo de +129 a –126 bytes a partir da instrução de desvio. Uma tentativa de usar a instrução BNE para um desvio maior resultará em um erro do tipo *out of range* (fora do alcance).

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: R

BPL (Branch on plus) Desvio, se positivo: Executa um desvio se o indicador N estiver desativado (isto é, se o resultado de um cálculo é positivo). Não resulta em nenhuma operação se o indicador N estiver ativado.

O destino do desvio é calculado pela adição de um deslocamento com sinal, variando entre –128 e +127, ao endereço da primeira instrução subseqüente à instrução BPL. Isto resulta num deslocamento efetivo de +129 a –126 bytes a partir da instrução de desvio. Uma tentativa de usar a instrução BPL para obter um desvio maior resultará em um erro do tipo *out of range* (fora do alcance).

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: R

* **BRA (Branch always)** Desvio sempre: Executa um desvio para um endereço específico de memória. A destinação do desvio é calculada pela adição de um deslocamento com sinal, variando entre −128 e +127, ao endereço da primeira instrução subseqüente à instrução BRA.

Este cálculo resulta num deslocamento efetivo de +129 bytes a −126 bytes a partir da instrução de desvio. Uma tentativa de utilização da instrução BRA para um desvio maior resultará num erro do tipo *out of range* (fora do alcance).

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: R

BRK (Break) A instrução BRK é uma interrupção (interrupt) controlada por software, geralmente utilizada para deter um programa, em determinado ponto, durante operações de depuração. Quando uma instrução BRK é encontrada em um programa, o contador de programa é incrementado em duas unidades e o indicador de break (B) do registro P é ativado. A seguir, o contador de programa é colocado na pilha (stack), com o byte superior em primeiro lugar. O conteúdo do registro P, com o indicador B ativado é, então, também colocado na pilha.

Quando estas operações são realizadas, o indicador de interrução I (interrupt flag) é ativado, desativando assim interrupções, e um endereço de 16 bits, armazenado em um ponteiro especial de 16 bits, é colocado no contador de programa. No Apple IIc e no Apple IIe, este ponteiro é contido nos endereços de memória $3F0 e $3F1, um par de registros destinados unicamente à utilização com a instrução BRK. Nos antigos computadores baseados no 6502, o ponteiro é contido nos endereços $FFFE e $FFFF – os mesmos endereços usados como um vetor por outros tipos de instruções de interrupção.

Quando uma instrução BRK é emitida, os ponteiros BRK usados pelos Apple IIc e IIe irão usualmente parar qualquer programa que está sendo executado e irão passar o controle do computador ao monitor de linguagem de máquina.

Quando o controle for devolvido ao monitor, os conteúdos dos registros de memória e dos registros do microprocessador podem ser examinados, e a depuração do programa pode prosseguir. Porém, como a instrução BRK produz uma série complexa de operações, seu uso pode, por vezes, apresentar resultados imprevisíveis. Para evitar surpresas desagradáveis quando da chamada de uma instrução BRK, o Apple IIc é equipado com um gerenciador de interrupções que pode manusear instruções BRK de várias formas, dependendo do computador estar no modo 80 colunas e da utilização da memória (auxiliar, chaveada).

Alterando o conteúdo do vetor BRK nos endereços de memória $3F0 e $3F1, os usuários do Apple IIc podem escrever suas próprias rotinas de manuseio da instrução BRK; mas, para a maioria das operações de depuração, provavelmente não haverá razão para tentar superar a rotina BRK residente do Apple IIc.

Indicadores afetados: B
Registros afetados: Ponteiro de pilha
Modos de endereçamento: IMP

BVC (Branch if overflow clear) Desvio se o overflow está desativado: Executa um desvio se o indicador de overflow V (Overflow flag) do registro P está desativado. Não resulta em nenhuma operação se o indicador V estiver ativado.

O destino do desvio é calculado pela adição de um deslocamento com sinal, variando entre −128 e +127, ao endereço da primeira instrução subseqüente à instrução BVC. Este cálculo resulta num deslocamento efetivo de +129 a −126 bytes a partir da instrução de desvio. Uma tentativa de utilização da instrução BVC para um desvio mais extenso resultará num erro *out of range* (fora do alcance). Esta instrução é, basicamente, utilizada em operações que envolvem números com sinais.

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: R

BVS (Branch if overflow set) Desvio se o indicador de overflow V (Overflow flag) estiver ativado. Não resulta em nenhuma operação se o indicador V estiver desativado. O destino do desvio é calculado pela adição de um deslocamento com sinal, variando de −128 a +127, ao endereço da primeira instrução subseqüente à instrução BVS.

Este cálculo resulta num deslocamento efetivo de +129 a −126 bytes a partir da instrução de desvio. Uma tentativa de utilização da instrução BVS para um desvio mais extenso resultará num erro *out of range* (fora do alcance). Esta instrução é, basicamente, utilizada em operações que envolvem números com sinais. Entretanto, ela também provê uma forma fácil de testar o bit 6 de qualquer valor.

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: R

CLC (Clear carry) Desativar transporte: Desativa o bit de transporte do registro de status do processador.

Indicadores afetados: C
Registros afetados: —
Modos de endereçamento: IMP

CLI (Clear interrupt mask) Desativar mascaramento de interrupção: Desativa o indicador de interrupção (interrupt flag) do registro P, permitindo a utilização de interrupções. Até o advento do Apple IIe, não havia suporte para as interrupções nos computadores série II da Apple, logo elas não eram utilizadas. Entretanto, o Apple IIe suporta interrupções, e o Apple IIc apresenta, inclusive, um gerenciador de interrupções extremamente sofisticado. As interrupções estão fora do escopo deste livro, mas vêm-se tornando dia a dia mais importantes, particularmente nos programas destinados à utilização de sofisticados periféricos, como por exemplo o mouse Apple. Detalhes a respeito da utilização das interrupções nos programas Apple IIc/IIe podem ser encontrados nos seus *reference manuals* e em manuais a respeito da operação de periféricos.

Indicadores afetados: I
Registros afetados: —
Modos de endereçamento: IMP

CLD (Clear decimal flag) Desativar o indicador de decimal: Desativa o indicador de decimal (decimal flag) do registro P, colocando o Apple IIc/IIe em modo binário (que é o modo default) em lugar do modo alternativo denominado BCD (ou modo decimal). Quando o Apple IIc está em modo binário, pode realizar operações binárias comuns em números binários comuns. Quando no modo BCD, pode trabalhar com um tipo especial de números, chamados números BCD, mais precisos que os números binários comuns, porém mais difíceis de usar. Os números BCD (abordados com mais detalhes nos Capítulos 2 e 10) são, geralmente, usados nos programas que lidam com o ramo dos negócios, onde é requerido um alto grau de precisão aritmética. Na maioria dos outros tipos de programas para o 6502B/65C02, o indicador de decimal é usualmente mantido desativado, utilizando-se os números binários comuns.

Indicadores afetados: D
Registros afetados: —
Modos de endereçamento: IMP

CLV (Cler overflow flag) Desativar o indicador de overflow: Desativa o indicador de overflow (overflow flag) do registro P. Esta instrução é usada, basicamente, em operações com números com sinal; entretanto, pode também ser utilizada para desativar o bit 6 de qualquer valor.

Indicadores afetados: V
Registros afetados: —
Modos de endereçamento: IMP

CMP (Compare with accumulator) Comparar com o acumulador: Compara qualquer número especificado, ou o conteúdo de uma localização especificada de memória, com o conteúdo do acumulador. Os indicadores N, Z e C do registro de status são afetados por esta operação, que é usualmente seguida por uma instrução de desvio (branch). Esta instrução pode causar o desvio para uma dada rotina, sob certas condições. Por exemplo, um desvio pode ocorrer se o valor do acumulador é menor que, igual a, ou maior que o valor que está sendo verificado. A instrução CMP e as instruções de desvio com elas utilizadas, são discutidas com mais detalhes no Capítulo 8.

Indicadores afetados: N, Z, C
Registros afetados: —
Modos de endereçamento: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

CPX (Compare with X register) Comparar com o registro X: Compara um número ou localização de memória especificados com o conteúdo do registro X. Os indicadores N, Z e C do registro de status são afetados por esta operação, que é usualmente seguida por uma instrução de desvio (branch). Esta instrução pode, sob certas condições, causar o desvio do programa para uma dada rotina. Por exemplo, um desvio pode ocorrer se o valor do registro X é menor, igual ou maior que o valor que está sendo verificado.

Indicadores afetados: N, Z, C
Registros afetados: —
Modos de endereçamento: A, IMM, Z

CPY (Compare with Y register) Comparar com o registro Y: Compara um número ou o conteúdo de uma localização especificada de memória com o conteúdo do registro Y. Os indicadores N, Z e C do registro de status são afetados por esta operação, que é usualmente seguida por uma instrução de desvio (branch), que pode causar o desvio do programa para uma dada rotina, sob certas condições. Por exemplo, o desvio poderá ocorrer se o valor do registro Y for menor que, igual a, ou maior que o valor que está sendo verificado.

Indicadores afetados: N, Z, C
Registros afetados: —
Modos de endereçamento: A, IMM, Z

DEA ou DEC A (Decrement accumulator) Decrementar acumulador: Decrementa unitariamente o conteúdo do acumulador. Se o valor do acumulador é $00, o resultado de uma operação DEA será $FF, uma vez que não há transporte.

Indicadores afetados: N, Z
Registros afetados: A
Modos de endereçamento: AC

DEC (Decrement a memory location) Decrementar uma localização de memória: Decrementa unitariamente o conteúdo de uma localização de memória especificada. Se o valor da localização é $00, o resultado de uma operação DEC será $FF, uma vez que não há transporte.

Indicadores afetados: N, Z
Registros afetados: M
Modos de endereçamento: ACC*, A, Z, AX, ZX

DEX (Decrement X register) Decrementar o registro X: Decrementa unitariamente o registro X. Se o valor do registro é $00, o resultado de uma operação DEX será $FF, uma vez que não há transporte.

Indicadores afetados: N, Z
Registros afetados: X
Modos de endereçamento: IMP

DEY (Decrement Y register) Decrementar o registro Y: Decrementa unitariamente o registro Y. Se o valor do registro é $00, o resultado de uma operação DEY será $FF, uma vez que não há transporte.

Indicadores afetados: N, Z
Registros afetados: Y
Modos de endereçamento: IMP

EOR (Exclusive-OR with accumulator) E exclusivo com o acumulador: Realiza uma operação E exclusiva com o conteúdo do acumulador e um valor ou localização de memória especificados. Os indicadores N e Z são condicionados de acordo com o resultado da operação, e o resultado é armazenado no acumulador.

Indicadores afetados: N, Z
Registros afetados: A
Modos de endereçamento: A, Z, I, AX, AY, IX, IY, ZX, ZPG*

INA ou INC A (Increment accumulator) Incrementar o acumulador: O conteúdo do acumulador é incrementado unitariamente. Se o conteúdo do acumulador é $FF, o resultado de uma operação INA será $00, uma vez que não há transporte.

Indicadores afetados: N, Z
Registros afetados: A
Modos de endereçamento: AC

INC (Increment memory) Incrementar memória: O conteúdo de uma localização de memória especificada é incrementado unitariamente. Se o valor da localização é $FF, o resultado de uma operação INC será $00, uma vez que não há transporte.

Indicadores afetados: N, Z
Registros afetados: M
Modos de endereçamento: ACC*, A, Z, AX, ZX

INX (Increment X register) Incrementar o registro X: O conteúdo do registro X é incrementado unitariamente. Se o conteúdo do registro é $FF, o resultado de uma operação INX será $00, uma vez que não há transporte.

Indicadores afetados: N, Z
Registros afetados: X
Modos de endereçamento: IMP

INY (Increment Y register) Incrementar o registro Y: O conteúdo do registro Y é in-

crementado unitariamente. Se o conteúdo do registro é $FF, o resultado de uma operação INY será $00, uma vez que não há transporte.

Indicadores afetados: N, Z
Registros afetados: Y
Modos de endereçamento: IMP

JMP (Jump to address) Pular para o endereço: Faz com que a execução do programa pule para um endereço especificado.

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: A, IX*, IND

JSR (Jump to subroutine) Pular para sub-rotina. Faz com que a execução do programa pule para o endereço que segue a instrução. Esse endereço deve ser o endereço de início de uma sub-rotina que termina com a instrução RTS. Quando o programa encontra essa instrução RTS, retorna à instrução subseqüente JSR que causou o salto para a sub-rotina.

Indicadores afetados: —
Registros afetados: Ponteiro de pilha
Modos de endereçamento: A

LDA (Load the accumulator) Carregar o acumulador: Carrega o acumulador com um valor ou localização de memória especificados. O indicador N será condicionado se um valor com o *high bit* ativado for carregado no acumulador, e o indicador Z será ativado se o valor carregado no acumulador for 0.

Indicadores afetados: N, Z
Registros afetados: A,
Modos de endereçamento: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

LDX (Load the X register) Carregar o registro X: Carrega o registro X com um valor ou localização de memória especificados. O indicador N será condicionado se um valor

com o *high bit* ativado for carregado no registro X, e o indicador Z será ativado se o valor carregado for 0.

Indicadores afetados: N, Z
Registros afetados: X
Modos de endereçamento: A, Z, IMM, AX, AY, IX, ZY

LDY (Load the Y register) Carregar o registro Y: Carrega o registro Y com um valor ou localização de memória especificados. O indicador N será condicionado se um valor com o *high bit* ativado for carregado no registro Y, e o indicador Z será ativado se o valor carregado for 0.

Indicadores afetados: N, Z
Registros afetados: Y
Modos de endereçamento: A, Z, IMM, AX, ZX

LSR (Logical shift right) Mudança de casa para a direita: Cada bit do acumulador é movido uma posição para a direita. Um 0 é depositado na posição do bit 7, e o bit 0 é depositado no transporte (carry). O resultado é deixado no acumulador ou no registro de memória afetado.

Indicadores afetados: N, Z, C
Registros afetados: A, M
Modos de endereçamento: AC, A, Z, AX, ZX

NOP (No operation) Nenhuma operação: Faz com que nada seja realizado pelo computador, durante dois ciclos de *clock*. Utiliza-se em atrasos (delay loops) e para sincronização de operações.

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: IMP

ORA (Inclusive-OR with the accumulator) OU inclusivo com o acumulador: Realiza

uma operação OU inclusivo binária com o valor do acumulador e um valor ou localização de memória especificados. Os indicadores Z e N são condicionados de acordo com o resultado da operação, e esse resultado é depositado no acumulador.

Indicadores afetados: N, Z
Registros afetados: A, M
Modos de endereçamento: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

PHA (Push accumulator) Empurrar o acumulador: O conteúdo do acumulador é "empurrado" para a pilha (stack). O acumulador e o registro P permanecem inalterados.

Indicadores afetados: —
Registros afetados: Ponteiro de pilha
Modos de endereçamento: IMP

PHP (Push processor status) Empurrar o status do processador: O conteúdo do registro P é "empurrado" para a pilha. Nenhum registro, incluindo o registro P, sofre alteração.

Indicadores afetados: —
Registros afetados: Ponteiro de pilha
Modos de endereçamento: IMP

*** PHX (Push X register on the stack)** Empurrar o registro X para a pilha: O conteúdo do registro X é "empurrado" para a pilha. Tanto o registro X como o registro P permanecem inalterados.

Indicadores afetados: —
Registros afetados: Ponteiro de pilha
Modos de endereçamento: IMP

*** PHY (Push Y register on the stack)** Empurrar o registro Y para a pilha: O conteúdo do registro Y é "empurrado" para a pilha. Tanto o registro Y quanto o registro P permanecem inalterados.

Indicadores afetados: —
Registros afetados: Ponteiro de pilha
Modos de endereçamento: IMP

PLA (Pull accumulator) Puxar para o acumulador: Um byte é removido da pilha (stack) e depositado no acumulador. Os indicadores N e Z são condicionados, exatamente como se houvesse sido realizada uma operação LDA.

Indicadores afetados: N, Z
Registros afetados: A, Ponteiro de pilha
Modos de endereçamento: IMP

PLP (Pull processor status) Puxar para o status do processador: Um byte é removido da pilha (stack) e depositado no registro P. Esta instrução é utilizada para recuperar a condição do registro P, após esta haver sido armazenada na pilha. Todos os indicadores são assim condicionados de forma a refletir o estado original do registro P.

Indicadores afetados: N, V, B, D, I, Z, C
Registros afetados: Ponteiro de pilha
Modos de endereçamento: IMP

*** PLX (Pull X register)** Puxar para o registro X: Um byte é removido da pilha e depositado no registro X. Os indicadores N e Z são condicionados, exatamente como se houvesse sido realizada uma operação LDX.

Indicadores afetados: N, Z
Registros afetados: X, Ponteiro de pilha
Modos de endereçamento: IMP

*** PLY (Pull Y register)** Puxar para o registro Y: Um byte é removido da pilha e depositado no registro Y. Os indicadores N e Z são condicionados, exatamente como se uma operação LDY houvesse sido realizada.

Indicadores afetados: N, Z
Registros afetados: Y, Ponteiro de pilha
Modos de endereçamento: IMP

ROL (Rotate left) Rodar para a esquerda: Cada bit do acumulador ou de uma localização de memória especificada é movido uma posição para a esquerda. O bit de transporte (carry bit) é depositado no lugar do bit 0 e é substituído pelo bit 7 do acumulador ou da localização de memória afetada. Os indicadores N e Z são condicionados de acordo com o resultado da operação de rotação.

Indicadores afetados: N, Z, C
Registros afetados: A, M
Modos de endereçamento: AC, A, Z, AX, ZX

ROR (Rotate right) Rodar para a direita: Cada bit do acumulador ou de uma localização de memória especificada é movido uma posição para a direta. O bit de transporte (carry bit) é depositado no lugar do bit 7 e é substituído pelo bit 0 do acumulador ou da localização de memória afetada. Os indicadores N e Z são condicionados de acordo com o resultado da operação de rotação.

Indicadores afetados: N, Z, C
Registros afetados: A, M
Modos de endereçamento: AC, A, Z, AX, ZX

RTI (Return from interrupt) Voltar da interrupção: A instrução RTI é utilizada para finalizar uma interrupção, de forma muito similar à utilizada com a instrução RTS para finalizar uma sub-rotina. Quando uma instrução RTI é emitida, o contador de programa e o registro P são retirados da pilha, e o ponteiro da pilha (stack pointer) é ajustado para refletir a nova condição da pilha.

A interrupção termina, e o programa pula para onde se encontrava antes da interrupção iniciar (isto é, para a instrução subseqüente à instrução que gerou a interrupção). Ao usar a instrução RTI, é importante lembrar que, embora ela restaure o registro P ao estado original, não restaura os registros X, Y e A; este trabalho é deixado a cargo do programador.

Indicadores afetados: N, V, B, D, I, Z, C
Registros afetados: PC, Ponteiro de pilha
Modos de endereçamento: IMP

RTS (Return from subroutine) Voltar da sub-rotina: A instrução RTS tem duas finalidades: quando usada no final de um programa em linguagem de máquina, ela termina o programa e passa o controle do Apple IIc/IIe a qualquer programa do sistema que estava em execução quando o programa em linguagem de máquina começou; usualmente, isto resulta num retorno ao BASIC ou ao monitor de linguagem de máquina.

Quando a RTS é utilizada no final de uma sub-rotina, tem uma função totalmente diferente: ela retira um endereço de 16 bytes do topo da pilha (stack) e carrega esse

endereço no contador de programa do 6502B/65C02. Esta operação termina a sub-rotina, e o programa em andamento pula para onde estava antes do início da sub-rotina (isto é, para a instrução subseqüente à instrução que chamou a sub-rotina).

Indicadores afetados: —
Registros afetados: Ponteiro de pilha
Modos de endereçamento: IMP

SBC (Subtract with carry) Subtrair com transporte: Subtrai um valor ou o conteúdo de uma localização de memória especificados, do conteúdo do acumulador. O oposto do transporte é também subtraído – em outras palavras, ocorre um empréstimo. Os indicadores N, V, Z e C são condicionados por esta operação, e o resultado da mesma é depositado no acumulador.

Indicadores afetados: N, V, Z, C
Registros afetados: A
Modos de endereçamento: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

SEC (Set carry) Ativar transporte: O indicador de transporte (carry flag) é ativado. Esta instrução usualmente precede uma instrução SBC. Sua finalidade básica é a de ativar o indicador de transporte, de forma que possa ocorrer um empréstimo.

Indicadores afetados: C
Registros afetados: —
Modos de endereçamento: IMP

SED (Set decimal mode) Ativar modo decimal: Prepara o computador para operações com números BCD (binary coded decimal = decimais codificados em binário). A aritmética BCD é mais precisa que a binária (o tipo normal de aritmética do 6502B/65C02), porém mais lenta, de utilização mais difícil, consumindo mais espaço de memória. A aritmética BCD é geralmente utilizada em programas financeiros e em operações aritméticas com ponto flutuante.

Indicadores afetados: D
Registros afetados: —
Modos de endereçamento: IMP

SEI (Set interrupt disable) Estabelecer desativamento de interrupções: Ativa o indica-

dor de interrupções I (interrupt flag) do registro P, desativando as interrupções mascaráveis (IRQ). As interrupções não-mascaráveis (NMI), essenciais à operação do Apple IIc/IIe, não são desativadas.

Indicadores afetados: I
Registros afetados: —
Modos de endereçamento: IMP

STA (Store accumulator) Armazenar o acumulador: Armazena o conteúdo do acumulador em uma localização de memória especificada. O conteúdo do acumulador não é afetado.

Indicadores afetados: —
Registros afetados: M
Modos de endereçamento: A, Z, AX, AY, IX, IY, ZX, ZPG*

STX (Store X register) Armazenar o registro X: Armazena o conteúdo do registro X em uma localização de memória especificada. O conteúdo do registro X não é afetado.

Indicadores afetados: —
Registros afetados: M
Modos de endereçamento: A, Z, ZY

STY (Store Y register) Armazenar o registro Y: Armazena o conteúdo do registro Y em uma localização de memória especificada. O conteúdo do registro Y não é afetado.

Indicadores afetados: —
Registros afetados: M
Modos de endereçamento: A, Z, ZX

*** STZ (Store 0 in memory)** Armazenar 0 na memória: Armazena um 0 em uma localização de memória especificada.

Indicadores afetados: —
Registros afetados: —
Modos de endereçamento: A, Z, AX, ZX

TAX (Transfer accumulator to X register) Transferir o acumulador para o registro X: O valor do acumulador é depositado para o registro X. Os indicadores N e Z são condicionados de acordo com o resultado desta operação. O conteúdo do acumulador não é alterado.

Indicadores afetados: N, Z
Registros afetados: X
Modos de endereçamento: IMP

TAY (Transfer accumulator to Y register) Transferir o acumulador para o registro Y: O valor do acumulador é depositado no registro Y. Os indicadores N e Z são condicionados de acordo com o resultado desta operação. O conteúdo do acumulador não é alterado.

Indicadores afetados: N, Z
Registros afetados: Y
Modos de endereçamento: IMP

*** TRB (Test and reset bits)** Testar e restabelecer bits: Uma operação lógica E é realizada com uma localização de memória e o inverso do valor do acumulador. O resultado é armazenado na localização de memória; o indicador Z é condicionado.

Indicadores afetados: D, Z, C, N
Registros afetados: M
Modos de endereçamento: A, Z

*** TSB (Test and set bits)** Testar e estabelecer bits: Uma operação lógica OU é realizada com uma localização de memória e o valor do acumulador. O resultado é armazenado na localização de memória e o indicador Z é condicionado.

Indicadores afetados: D, Z, C, N
Registros afetados: M
Modos de endereçamento: A, Z

TSX (Transfer stack to X register) Transferir da pilha para o registro X: O valor do ponteiro da pilha (stack pointer) é depositado no registro X. Os indicadores N e Z são condicionados de acordo com o resultado desta operação. O valor do ponteiro da pilha não é alterado.

Indicadores afetados: N, Z
Registros afetados: X
Modos de endereçamento: IMP

TXA (Transfer X register to accumulator) Transferir o registro X para o acumulador: O valor do registro X é depositado no acumulador. Os indicadores N e Z são condicionados de acordo com o resultado desta operação. O valor do registro X não é alterado.

Indicadores afetados: N, Z
Registros afetados: A
Modos de endereçamento: IMP

TXS (Transfer X register to stack) Transferir o registro X para a pilha: O valor do registro X é depositado no ponteiro da pilha (stack pointer). Nenhum indicador é condicionado por esta operação. O valor do registro X não é alterado.

Indicadores afetados: —
Registros afetados: Ponteiro de pilha
Modos de endereçamento: IMP

TYA (Transfer Y register to accumulator) Transferir o registro Y para o acumulador: O valor presente no registro Y é depositado no acumulador. Os indicadores N e Z são condicionados por esta operação. O valor do registro Y não é alterado.

Indicadores afetados: N, Z
Registros afetados: A
Modos de endereçamento: IMP

O SIMULADOR DE BYTES

O Programa 6-1 é uma listagem do "Simulador de Bytes", um programa que lhe permitirá ver o que o microprocessador de seu computador executa quando processa as instruções acima.

O Simulador de Bytes não é um assembler real; não poderá ser usado para compilar e rodar programas. Ele apresenta, também, mais algumas limitações: por exemplo,

não permite a utilização de rótulos (labels) ou endereçamento indireto (tópico que será abordado em detalhes num capítulo mais adiante). E, como o programa foi elaborado de forma a ser compatível tanto com o Apple IIe quanto com o Apple IIc, não aceitará as oito instruções exclusivas do 65C02.

Uma característica importante do Simulador de Bytes é a capacidade de congelar o funcionamento do computador, fazendo com que o sistema pare em cada passo da execução. O Simulador de Bytes não introduzirá valores nos registros de memória do computador; ele pode *ler* o conteúdo de qualquer registro de memória, mas não pode *gravar* em RAM.

Desta forma, pode-se utilizar o programa para verificar o conteúdo de qualquer registro de memória, sem causar um desastre cada vez que digitar uma instrução em linguagem assembly.

Após haver digitado e gravado o Simulador de Bytes, poderemos apreciar seu funcionamento, usando-o para digitar qualquer programa apresentado até agora neste livro. Uma vez entendido o funcionamento, poderemos usar o Simulador de Bytes para testar qualquer programa em linguagem assembly para os Apple IIe e IIc.

Ao carregar o programa, inicialmente veremos uma linha de status ao longo da parte superior da tela. Esta linha mostrará a condição atual dos quatro registros internos mais importantes do 6502B/65C02: o registro de status do processador, o acumulador e os registros X e Y. Sob a linha de status, haverá um cursor. Partindo da posição do cursor, podemos digitar um programa em assembly e ver exatamente como cada instrução desse programa afetará os registros do microprocessador, como se o programa estivesse realmente rodando.

A cada linha digitada, o Simulador de Bytes mostrará como essa linha afetaria os registros internos da CPU, e manterá uma atualização do conteúdo dos registros A, X e Y. A informação será apresentada em ambas as notações, hexa e binária.

O Simulador de Bytes deverá ser de grande valia em seu aprendizado da linguagem assembly 6502B/65C02, permitindo um acompanhamento dos efeitos de cada instrução em assembly nos registros internos da CPU.

Programa 6-1
O SIMULADOR DE BYTES

```
100  REM  ********************  O SIMULADOR DE BYTES  **********
     **********
110  DATA   ASL,BRK,CLC,CLD,CLI,CLV,DEX,DEY,INX,INY,LSR,NOP,
     PHA,PHP,PLA
```

```
120  DATA  PLP,ROL,ROR,RTI,RTS,SEC,SED,SEI,TAX,TAY,TSX,TXA,
     TXS,TYA
130  DATA  ADC,AND,CMP,CPX,CPY,EOR,LDA,LDX,LDY,ORA,SBC
140  DATA  ADC,AND,ASL,BCC,BCS,BEQ,BIT,BMI,BNE,BPL,BVC,BVS,
     CMP,CPX,CPY
150  DATA  DEC,EOR,INC,JMP,JSR,LDA,LDX,LDY,LSR,ORA,ROL,ROR,
     SBC,STA,STX
160  DATA  STY,ASL,LSR,ROL,ROR
170  DATA  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
180  DATA  0000,0001,0010,0011,0100,0101,0110,0111
190  DATA  1000,1001,1010,1011,1100,1101,1110,1111
200  DIM HEX$(8),BIT$(8),H$(16),B$(16),TEMP$(2),BIT(8),N$(75)
210 AH$ = "00":XH$ = "00":YH$ = "00":Z$ = "0":AD = 0:SC =
     0:YD = 0
220 AB$ = "00000000":XB$ = "00000000":YB$ = "00000000"
230 N = 0:V = 0:B = 0:D = 0:I = 0:Z = 0:C = 0
240  IF  RIGHT$ (A$,1) = " " THEN 300
250  FOR L = 1 TO 75: READ N$(L): NEXT L
260  FOR L = 1 TO 16: READ H$(L): NEXT L: FOR L = 1 TO 16:
     READ B$(L): NEXT L
270  PRINT (CHR$(4); "PR#3": TEXT : REM LIMPAR TELA,
     INICIAR PROGRAMA
280  PRINT "NV-BDIZC   A: ";AH$;"       X: ";XH$;"       Y: "
     ;YH$
290  PRINT N;V;"-";B;D;I;Z;C;"   ";AB$;"   ";XB$;"   ";YB$:
     PRINT : GOTO 310
300  PRINT (CH$(7): REM 'BIP'
310  B=0 : A$ = " ": INPUT " "; A$ : REM TECLE ESPAÇO ENTRE AS ASPAS
     APOS O INPUT
320  IF  LEN (A$) < 3 OR  LEN (A$) > 10 THEN 300
330  IF LEN (A$)<3 THEN 420 : REM IR PARA AS ROTINAS DE MNEMONICOS
     DE UM BYTE
340  IF  MID$ (A$,4,1) <  >  CHR$ (32) THEN 300
350  IF  RIGHT$ (A$,1) =  CHR$ (32) THEN 300
360  GOTO 450 : REM IR PARA AS ROTINAS DE MNEMONICOS DE BYTES MULTIPLOS
370  REM *** ROTINA PARA CONVERTER OP$ E AD$ EM NUMEROS
     BINARIOS  ******
380 OD$ = OP$: GOSUB 1130: FOR L = 1 TO 8:BIT$(L) = Z$:
     NEXT L
390  FOR L = 1 TO 8:B1$(L) =  MID$ (OB$,L,1): NEXT L
400  FOR L = 1 TO 8:B2$(L) =  MID$ (AB$,L,1): NEXT L: RETURN
410  REM  ************** ENDERECAMENTO IMPLICITO ****************
     *********
420  FOR L = 1 TO 29: IF A$ = N$(L) THEN OC$ = A$:OC = L:
     GOTO 1350
430  NEXT L: GOTO 300
440  REM  ****  ENDEREÇAMENTOS IMEDIATO, ABSOLUTO E DO ACUMULADOR
     *********
450 OC$ =  LEFT$ (A$,3):OP$ =  MID$ (A$,5)
```



```
460  IF  LEFT$ (OP$,1) = "#" AND  MID$ (OP$,2,1) = "$" THEN
     FLAG$ = "AH": GOTO 530
470  IF  LEFT$ (OP$,1) = "#" THEN FLAG$ = "AD": GOTO 620
480  IF  LEFT$ (OP$,1) = "$" THEN FLAG$ = "IH": GOTO 690
490  IF OP$ = "A" THEN 790
500  IF  LEFT$ (OP$,1) < "0" AND  LEFT$ (OP$,1) > "9" THEN
     300: REM   TENTAR NOVAMENTE
510 FLAG$ = "ID": GOTO 840
520  REM  ********* OPERANDO HEXA, ENDERECAMENTO ABSOLUTO ******
     **************
530 OP$ =  MID$ (OP$,3):BA$ = "H": REM  ENDERECO HEXA
540  FOR L = 30 TO 40: IF OC$ = N$(L) THEN OC = L: GOTO 560
550  NEXT L: GOTO 300
560  IF  LEN (OP$) > 2 THEN 300
570  FOR L = 1 TO  LEN (OP$):X$ =  MID$ (OP$,L,1): IF X$ <
     Z$ OR X$ > "F" THEN 300
580  IF X$ > "9" AND X$ < "A" THEN 300
590  IF  LEN (OP$) = 1 THEN OP$ = Z$ + OP$
600 OH$ = OP$: GOSUB 1030:OP$ = OD$: GOTO 670
610  REM  *********  OPERANDO DECIMAL, ENDERECAMENTO ABSOLUTO  **
     ************
620 OP$ =  MID$ (OP$,2):BA$ = "D": REM  ENDERECO DECIMAL
630  FOR L = 30 TO 40: IF OC$ = N$(L) THEN OC = L: GOTO 650
640  NEXT L: GOTO 300
650  IF  VAL (OP$) > 255 THEN 300
660  FOR L = 1 TO  LEN (OP$):X$ =  MID$ (OP$,L,1): IF  ASC
     (X$) < 48 OR  ASC
(X$) > 57 THEN 420
670 OC = OC - 29: GOTO 1420
680  REM  ********* OPERANDO HEXA,  ENDERECAMENTO IMEDIATO  *****
     ***********
690 OP$ =  MID$ (OP$,2):BA$ = "H": REM  ENDERECO HEXA
700  FOR L = 41 TO 71: IF OC$ = N$(L) THEN OC = L: GOTO 720
710  NEXT L: GOTO 300
720  IF  LEN (OP$) > 4 THEN 300
730  FOR L = 1 TO  LEN (OP$):X$ =  MID$ (OP$,L,1): IF X$ <
     Z$ OR X$ > "F" THEN 300
740  IF X$ > "9" AND X$ < "A" THEN 300
750  NEXT L
760 OH$ = OP$: GOSUB 1030:OP$ = OD$:OP$ =  STR$ ( PEEK ( VAL
     (OP$)))
770 BA$ = "D":OC = OC - 40: GOTO 1430
780  REM  ************** ENDERECAMENTO DO ACUMULADOR *********
     ***********
790 OP$ = "A": REM  ENDERECAMENTO DO ACUMULADOR
800  FOR L = 72 TO 75: IF OC$ = N$(L) THEN OC = L - 71: GOTO
     820
810  NEXT L: GOTO 300
820  ON OC GOTO 1510,1760,1790,1810
```

```
830  REM  ********** OPERANDO DECIMAL, ENDERECAMENTO IMEDIATO
     **********
840 BA$ = "D": REM  ENDERECO DECIMAL
850  IF  VAL (OP$) > 65535 THEN 300
860  FOR L = 1 TO  LEN (OP$):X$ =  MID$ (OP$,L,1): IF  ASC
     (X$) < 48 OR  ASC
(X$) > 57 THEN 300
870  FOR L = 41 TO 71: IF OC$ = N$(L) THEN OC = L: GOTO 890
880  NEXT L: GOTO 300
890 OP$ =  STR$ ( PEEK ( VAL (OP$)))
900 OC = OC - 40: GOTO 1430
910  REM  ********** CONVERSAO DECIMAL PARA HEXADECIMAL **
     **********
920  FOR L = 1 TO 4:HEX$(L) = "": NEXT L
930  FOR L = 1 TO 5:T$ =  RIGHT$ (OD$,L): NEXT L
940 NR =  VAL (OD$):X = 4
950 TMP = NR:NR =  INT (NR / 16):TMP = TMP - NR * 16
960  IF TMP < 10 THEN HEX$(X) =  RIGHT$ ( STR$ (TMP),1): GOTO 980
970 HEX$(X) =  CHR$ (TMP - 10 +  ASC ("A"))
980  IF NR <  > 0 THEN X = X - 1: GOTO 950
990 OH$ = HEX$(1) + HEX$(2) + HEX$(3) + HEX$(4)
1000  IF  LEN (OH$) = 1 THEN OH$ = Z$ + OH$
1010  RETURN
1020  REM  ********** CONVERSAO HEXADECIMAL PARA DECIMAL  ***
      **********
1030 NR = 0: FOR L = 1 TO  LEN (OH$):HEX$(L) =  MID$
     (OH$,L,1)
1040  IF HEX$(L) <  = "9" THEN NR = NR * 16 +  VAL (HEX$(L)):
     GOTO 1060
1050 NR = NR * 16 +  ASC (HEX$(L)) -  ASC ("A") + 10
1060  NEXT L:OD$ =  STR$ (NR): RETURN
1070  REM  ********** CONVERSAO BINARIO PARA DECIMAL  ****
      **********
1080  FOR L = 8 TO 1 STEP  - 1:B$(L) =  MID$ (OB$,L,1):
     NEXT L
1090  FOR L = 1 TO 8:BIT(L) =  VAL (B$(L)): NEXT L:OD =
     0:M = 256
1100  FOR L = 1 TO 8:M = M / 2:OD = OD + BIT(L) * M: NEXT L
1110 OD$ =  STR$ (OD): RETURN
1120  REM  ********** CONVERSAO DECIMAL PARA BINARIO  ********
      **********
1130 OD =  VAL (OP$): FOR L = 8 TO 1 STEP  - 1:Q = OD / 2:R
     = Q -  INT (Q)
1140  IF R = 0 THEN BT$(L) = Z$: GOTO 1160
1150 BT$(L) = "1"
1160 OD =  INT (Q): NEXT L
1170 OB$ = BT$(1) + BT$(2) + BT$(3) + BT$(4) + BT$(5) +
     BT$(6) + BT$(7) + BT$(8): RETURN
1180  REM  ******** CONVERSAO HEXADECIMAL PARA BINARIO  ******
      **********
```



```
1190 HEX$(1) = "":HEX$(2) = "": FOR L = 1 TO  LEN (OH$):
     HEX$(L) =  MID$ (OH$,L,1)
1200  NEXT L: IF HEX$(2) = "" THEN HEX$(2) = HEX$(1):HEX$(1)
     = "0"
1210  FOR L = 1 TO 16: IF HEX$(1) = H$(L) THEN BIT$(1) =
     B$(L)
1220  NEXT L
1230  FOR L = 1 TO 16: IF HEX$(2) = H$(L) THEN BIT$(2) =
     B$(L)
1240  NEXT L
1250 OB$ = BIT$(1) + BIT$(2): PRINT : RETURN
1260  REM  ********** CONVERSAO BINARIO PARA HEXADECIMAL  **
      **********
1270  FOR L = 1 TO 8:BIT$(L) =  MID$ (OB$,L,1): NEXT L
1280 BIT$ = BIT$(1) + BIT$(2) + BIT$(3) + BIT$(4) + BIT$(5)
     + BIT$(6) + BIT$(7) + BIT$(8)
1290 T1$ =  LEFT$ (BIT$,4):T2$ =  RIGHT$ (BIT$,4): FOR L =
     1 TO 16
1300  IF T1$ = B$(L) THEN HEX$(1) = H$(L)
1310  NEXT L: FOR L = 1 TO 16: IF T2$ = B$(L) THEN HEX$(2) =
     H$(L)
1320  NEXT L: IF HEX$(1) = "" THEN HEX$(1) = Z$: IF HEX$(2)
     = "" THEN HEX$(2) = Z$
1330 OH$ = HEX$(1) + HEX$(2): RETURN
1340  REM  ********** DADOS ON/GOTO **********
      **********
1350  ON OC GOTO 1510,1560,1570,1580,1590,1600,1610,1650,
     1690,1730
1360 NR = OC - 10
1370  ON NR GOTO 1760,1780,1780,1780,1780,1780,1790,1810,
     1830,1840
1380 NR = NR - 10
1390  ON NR GOTO 1850,1860,1870,1880,1900,1920,1930,1940,
     1950,1840
1400 NR = NR - 10
1410  ON NR GOTO 1850,1860,1870,1880,1900,1920,1930,1940,
     1950,1950
1420  ON OC GOTO 1970,2100,2170,2250,2320,2390,2450,2550,
     2650,2750,2810
1430  ON OC GOTO 1970,2350,3250,1780,1780,1780,3200,1780,
     1780,1780
1440 NR = OC - 10
1450  ON NR GOTO 1780,1780,3250,2250,2320,3260,2390,3290,
     1780,1780
1460 NR = NR - 10
1470  ON NR GOTO 2450,2550,2650,3320,2750,3340,3360,2810,
     1780,1780
1480 NR = NR - 10
1490  ON NR GOTO 1780,3390
```

```
1500  REM  ********** ROTINAS OP-CODE INICIAM AQUI **********
     ***
1510 C =  VAL ( LEFT$ (AB$,1)):AB$ =  MID$ (AB$,2) + Z$:
     REM  *** ASL ***
1520 OB$ = AB$: GOSUB 1270:AH$ = OH$:OP$ = OH$: GOSUB 1030:N
     = 0:Z = 0
1530  IF  LEFT$ (OB$,1) = "1" THEN N = 1
1540  IF  VAL (OD$) = 0 THEN Z = 1
1550  GOTO 3390
1560 B = 1: GOTO 3390: REM  *** BRK ***
1570 C = 0: GOTO 3390: REM  *** CLCL ***
1580 D = 0: GOTO 3390: REM  *** CLD ***
1590 I = 0: GOTO 3390: REM  *** CLI ***
1600 V = 0: GOTO 3390: REM  *** CLV ***
1610 OH$ = XH$: GOSUB 1030:XD =  VAL (OD$): REM  *** DEX ***
1620 XD = XD - 1: IF XD < 0 THEN XD = 255
1630 OD$ =  STR$ (XD): GOSUB 920:XH$ = OH$: GOSUB 1190:XB$
     = OB$
1640 TMP = XD: GOSUB 3410: GOTO 280
1650 OH$ = YH$: GOSUB 1030:YD =  VAL (OD$): REM  *** DEY ***
1660 YD = YD - 1: IF YD < 0 THEN YD = 255
1670 OD$ =  STR$ (YD): GOSUB 920:YH$ = OH$: GOSUB 1190:YB$
     = OB$
1680 TMP = YD: GOSUB 3410: GOTO 280
1690 OH$ = XH$: GOSUB 1030:XD =  VAL (OD$): REM  *** INX ***
1700 OD$ =  STR$ (XD): GOSUB 920:XH$ = OH$: GOSUB 1190:XB$ =
     OB$
1710 XD = XD + 1: IF XD > 255 THEN XD = 0
1720  GOTO 1630
1730 OH$ = YH$: GOSUB 1030:YD =  VAL (OD$): REM  *** INY ***
1740 YD = YD + 1: IF YD > 255 THEN YD = 0
1750  GOTO 1670
1760 C =  VAL ( RIGHT$ (AB$,1)):AB$ = Z$ +  LEFT$ (AB$,7):
     REM  *** LSR ***
1770  GOTO 1520
1780  GOTO 3390: REM  *** NOP, PHA, PHP, PLA  E  PLP ***
1790 TMP = C:C =  VAL ( LEFT$ (AB$,1)):AB$ =  MID$ (AB$,2) +
     STR$ (TMP): REM
*** ROL ***
1800  GOTO 1520
1810 TMP = C:C =  VAL ( RIGHT$ (AB$,1)):AB$ =  STR$ (TMP) +
     LEFT$ (AB$,7):
REM  *** ROR ***
1820  GOTO 1520
1830 N = 0:V = 0:B = 0:D = 0:I = 0:Z = 0:C = 0: GOTO 280:
     REM  *** RTI ***
1840  GOTO 3390: REM  *** RTS ***
1850 C = 1: GOTO 3390: REM  *** SEC ***
1860 D = 1: GOTO 3390: REM  *** SED ***
```



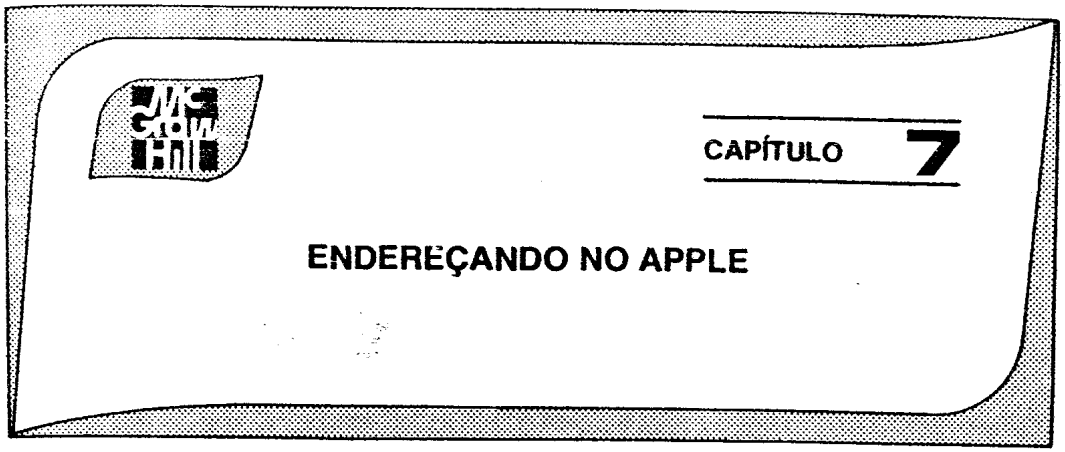
```
1870 I = 1: GOTO 3390: REM  *** SEI ***
1880 XH$ = AH$:XB$ = AB$:OP$ = AH$: GOSUB 1030:TMP =  VAL
     (OD$): REM
***              TAX ***
1890  GOSUB 3410: GOTO 3390
1900 YH$ = AH$:YB$ = AB$:OP$ = AH$: GOSUB 1030:TMP =  VAL
     (OD$): REM  *** TAY
***
1910  GOSUB 3410: GOTO 3390
1920 XH$ = "00":XB$ = "00000000": GOSUB 3410: GOTO 3390: REM
     *** TSX ***
1930 AH$ = XH$:AB$ = XB$: GOTO 1520: REM    *** TXA ***
1940  GOTO 3390: REM  *** TXS ***
1950 AH$ = YH$:AB$ = YB$: GOTO 1520: REM   *** TYA ***
1960  REM  ********** OPERANDOS DE ENDERECO ABSOLUTO *******
     *******
1970  IF D THEN 2950: REM  *** ADC ***
1980 OP =  VAL (OP$):TMP$ = AB$
1990  GOSUB 1130:PLUS$ = OB$
2000 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$):TMP = AD
2010 AD = AD + OP + C:C = 0: IF AD > 255 THEN  GOSUB 2090
2020 AD$ =  STR$ (AD):OD$ = AD$: GOSUB 920:AH$ = OH$
2030  GOSUB 1190:AB$ = OB$
2040 N = 0: IF AD > 127 THEN N = 1
2050 Z = 0: IF AD = 0 THEN Z = 1
2060 V = 0
2070  IF  LEFT$ (TMP$,1) =  LEFT$ (PLUS$,1) AND  LEFT$
     (TMP$,1) <  >  LEFT$ (AB$,1) THEN V = 1
2080 OD$ = AD$: GOSUB 1030:AH$ = OH$: GOTO 280
2090 C = 1:AD = AD - 256: RETURN
2100  GOSUB 380: REM   *** AND ***
2110  FOR L = 1 TO 8:BIT$(L) = "0": NEXT L
2120  FOR L = 1 TO 8: IF B1$(L) = B2$(L) THEN BIT$(L) =
     B1$(L)
2130  NEXT L
2140 AB$ = BIT$(1) + BIT$(2) + BIT$(3) + BIT$(4) + BIT$(5) +
     BIT$(6) + BIT$(7) + BIT$(8)
2150 OB$ = AB$: GOSUB 1270:AH$ = OH$
2160  GOSUB 1030:TMP =  VAL (OD$): GOSUB 3410: PRINT :
     GOTO 280
2170 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$):OP =
     VAL (OP$): REM  *** CMP ***
2180  IF AD = OP THEN Z = 1: GOTO 2200
2190 Z = 0
2200  IF OP > AD THEN N = 1: GOTO 2220
2210 N = 0
2220  IF AD > OP OR AD = OP THEN C = 1: GOTO 2240
2230 C = 0
2240  GOTO 3390
2250 OH$ = XH$: GOSUB 1030:XD$ = OD$:XD =  VAL (XD$):OP =
```

```
    VAL (OP$): REM  CPX
2260  IF XD = OP THEN Z = 1: GOTO 2280
2270 Z = 0
2280  IF OP > XD THEN N = 1: GOTO 2300
2290 N = 0
2300  IF XD > OP OR XD = OP THEN C = 1: GOTO 2320
2310 C = 0: GOTO 3380
2320 OH$ = YH$: GOSUB 1030:YD$ = OD$:YD =  VAL (YD$):OP =
    VAL (OP$): REM  *** CPY ***
2330  IF YD = OP THEN Z = 1: GOTO 2350
2340 Z = 0
2350  IF OP > YD THEN N = 1: GOTO 2370
2360 N = 0
2370  IF YD > OP OR YD = OP THEN C = 1: GOTO 3110
2380 C = 0: GOTO 3380
2390  GOSUB 400: REM  *** EOR ***
2400  FOR L = 1 TO 8:BIT$(L) = "0": NEXT L: FOR L = 1 TO 8
2410  IF B1$(L) = "1" OR B2$(L) = "1" THEN 2440
2420  IF B1$(L) = B2$(L) THEN 2440
2430 BIT$(L) = "1"
2440  NEXT L: GOTO 2140
2450  IF D = 1 THEN 2480: REM   *** LDA ***
2460 OD$ = OP$: GOSUB 920:AH$ = OH$: GOSUB 1190:AB$ = OB$
2470 TMP =  VAL (OP$): GOSUB 3410: GOTO 280
2480  IF FLAG$ = "AD" AND  VAL (OP$) > 99 THEN 300
2490  IF FLAG$ <  > "AD" THEN OD$ = OP$: GOSUB 920:AH$ =
    OH$: GOTO 2530
2500  IF  LEN (OP$) = 1 THEN OP$ = Z$ + OP$
2510 AH$ = OP$:OH$ = AH$: GOTO 2530
2520 OD$ = OP$: GOSUB 920:AH$ = OH$
2530  GOSUB 1190:AB$ = OB$
2540 TMP =  VAL (OP$): GOSUB 3410: GOTO 2470
2550  IF D = 1 THEN 2580: REM    *** LDX ***
2560 OD$ = OP$: GOSUB 920:XH$ = OH$: GOSUB 1190:XB$ = OB$
2570 TMP =  VAL (OP$): GOSUB 3410: GOTO 280
2580  IF FLAG$ = "AD" AND  VAL (OP$) > 99 THEN 300
2590  IF FLAG$ <  > "AD" THEN OD$ = OP$: GOSUB 920:XH$ =
    OH$: GOTO 2630
2600  IF  LEN (OP$) = 1 THEN OP$ = Z$ + OP$
2610 XH$ = OP$:OH$ = XH$: GOTO 2630
2620 OD$ = OP$: GOSUB 920:XH$ = OH$
2630  GOSUB 1190:XB$ = OB$
2640 TMP =  VAL (OD$): GOSUB 3410: GOTO 2570
2650  IF D = 1 THEN 2680: REM  *** LDY ***
2660 OD$ = OP$: GOSUB 920:YH$ = OH$: GOSUB 1190:YB$ = OB$
2670 TMP =  VAL (OP$): GOSUB 3410: GOTO 280
2680  IF FLAG$ = "AD" AND  VAL (OP$) > 99 THEN 300
2690  IF FLAG$ <  > "AD" THEN OD$ = OP$: GOSUB 920:YH$ =
    OH$: GOTO 2730
```



```
2700  IF  LEN (OP$) = 1 THEN OP$ = Z$ + OP$
2710 YH$ = OP$:OH$ = YH$: GOTO 2730
2720 OD$ = OP$: GOSUB 920:YH$ = OH$
2730  GOSUB 1190:YB$ = OB$
2740 TMP =  VAL (OD$): GOSUB 3410: GOTO 2670
2750  GOSUB 380: REM  *** ORA ***
2760  FOR L = 1 TO 8: IF B1$(L) = "1" OR B2$(L) = "1" THEN
    BIT$(L) = "1"
2770  NEXT L
2780 AB$ = "": FOR L = 1 TO 8:AB$ = AB$ + BIT$(L): NEXT L
2790 OB$ = AB$: GOSUB 1270:AH$ = OH$
2800  GOSUB 1030:TMP =  VAL (OD$): GOSUB 3410: GOTO 3390
2810  IF D THEN 3060: REM  *** SBC ***
2820 OP =  VAL (OP$):TMP$ = AB$
2830  GOSUB 1130:MI$ = OB$
2840 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$):TMP = AD
2850 AD = AD - OP: IF C = 0 THEN AD = AD - 1
2860  IF AD < 0 THEN AD = 256 + AD:C = 0
2870 AD$ =  STR$ (AD):OD$ = AD$: GOSUB 920:AH$ = OH$
2880  GOSUB 1190:AB$ = OB$
2890 N = 0: IF AD > 127 THEN N = 1
2900 Z = 0: IF AD = 0 THEN Z = 1
2910 V = 0: IF  LEFT$ (TMP$,1) =  LEFT$ (MI$,1) THEN 2930
2920  IF  LEFT$ (AB$,1) =  LEFT$ (TMP$,1) THEN V = 1
2930 OD$ = AD$: GOSUB 1030:AH$ = OH$: GOTO 280
2940  REM   ***** ROTINA DE ADICAO BCD *****
2950  IF FLAG$ <  > "AD" THEN 1980
2960  IF  LEFT$ (AH$,1) > "9" OR  RIGHT$ (AH$,1) > '9" THEN
    3030
2970 AD =  VAL (AH$)
2980 OP =  VAL (OP$):AD = AD + OP + C:C = 0
2990  GOSUB 1030:TMP =  VAL (OD$): GOSUB 3410: GOTO 3390
3000  IF AD > 99 THEN  GOSUB 3040
3010 AH$ =  STR$ (AD): IF  LEN (AH$) = 1 THEN AH$ = Z$ + AH$
3020 OH$ = AH$: GOSUB 1190:AB$ = OB$: GOTO 280
3030 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$): GOTO
    2980
3040 C = 1:AD = AD - 100: RETURN
3050  REM  ******* ROTINA DE SUBTRACAO BCD ********************
    **
3060  IF FLAG$ <  > "AD" THEN 2820
3070  IF  LEFT$ (AH$,1) > "9" OR  RIGHT$ (AH$,1) > "9" THEN
    3120
3080 AD =  VAL (AH$)
3090 OP =  VAL (OP$):AD = AD - OP: IF C = 0 THEN AD = AD - 1
3100  IF AD < 0 THEN  GOSUB 3130
3110  GOTO 3010
3120 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$): GOTO
    3090
3130 C = 0:AD = 100 + AD: RETURN
```

```
3140  REM  ******** OPERANDOS DE ENDERECO IMEDIATO  ************
     ****
3150 OD$ = OP$: GOSUB 920: GOSUB 1190: REM  ASL
3160 C =  VAL ( LEFT$ (OB$,1)):OB$ =  MID$ (OB$,2) + Z$
3170  GOSUB 1270: GOSUB 1030:X =  VAL (OD$): IF X < 0 THEN N
3180  IF X = 0 THEN Z = 1
3190  GOTO 280
3200  GOSUB 1030:AD =  VAL (OD$): REM    *** BIT ***
3210  GOSUB 1130:N =  VAL ( LEFT$ (OB$,1)):V =  VAL ( MID$
     (OB$,2,1))
3220  GOSUB 400:Z = 1: FOR L = 1 TO 8
3230  IF B1$(L) = "1" AND B2$(L) = "1" THEN Z = 0
3240  NEXT L: GOTO 3390
3250  GOTO 2170
3260 OP =  VAL (OP$):OP = OP - 1: IF OP < 0 THEN N: REM  DEC
3270  IF OP = 0 THEN Z = 1
3280  GOTO 3390
3290 OP =  VAL (OP$):OP = OP + 1: IF OP < 0 THEN N: REM  INC
3300  IF OP = Z THEN Z
3310  GOTO 3390
3320 OD$ = OP$: GOSUB 920: GOSUB 1190
3330 C =  VAL ( RIGHT$ (OB$,1)):OB$ = Z$ +  LEFT$ (OB$,7):
     GOTO 3170
3340 OD$ = OP$: GOSUB 920: GOSUB 1190: REM  ROL
3350 TMP = C:C =  VAL ( LEFT$ (OB$,1)):OB$ =  MID$ (OB$,2) +
     STR$ (TMP): GOTO 3170
3360 OD$ = OP$: GOSUB 920: GOSUB 1190: REM  ROR
3370 TMP = C:C =  VAL ( RIGHT$ (OB$,1)):OB$ =  STR$ (TMP) +
     LEFT$ (OB$,7): GOTO 3170
3380  REM  ********** COLOCAR UMA LINHA EM BRANCO E TOMAR OUTRA LINHA
3390  PRINT : GOTO 280
3400  REM  ************** ATIVAR INDICADORES Z E N ***************
     ***
3410 N = 0: IF TMP > 127 THEN N = 1
3420 Z = 0: IF TMP = 0 THEN Z = 1
3430  RETURN
3440  IF FLAG$ = "AH" AND ( LEFT$ (AH$,1)) < "A" AND ( MID$
     (AH$,2,1)) < "A" THEN 2480
```



CAPÍTULO **7**

ENDEREÇANDO NO APPLE

No Capítulo 1, dissemos que existe uma relação direta entre linguagem assembly e linguagem de máquina: para cada mnemônico em um programa em assembly há uma instrução numérica em linguagem de máquina, que tem exatamente o mesmo significado.

Na realidade, existem vários mnemônicos assembly que possuem mais de uma instrução equivalente em linguagem de máquina. Por exemplo, o mnemônico ADC pode ser convertido em oito diferentes instruções numéricas, ao ser transformado em linguagem de máquina. Para compreendermos o porquê, é necessário saber algo a respeito dos modos de endereçamento na linguagem assembly do 6502B/65C02.

Um modo de endereçamento é uma técnica para localizar e utilizar a informação armazenada na memória do computador. O microprocessador 6502 tem treze modos de endereçamento, e o 65C02 apresenta quinze modos. Assim, o Apple IIc ou IIe terá treze ou quinze modos de endereçamento, dependendo do processador instalado. Neste capítulo, iremos examinar os quinze modos de endereçamento.

Vejamos, primeiramente, a Tabela 7-1, que mostra que existem nove modos de endereçamento que podem ser usados com o mnemônico ADC.

Examinando as Colunas 2 e 3 da Tabela 7-1, notamos uma curiosa relação entre os comandos em linguagem assembly, da Coluna 2, e seus equivalentes em linguagem de máquina, da Coluna 3. Na Coluna 2, os modos de endereçamento usam o mesmo mnemônico, mas um operando diferente; na Coluna 3, os comandos têm o mesmo operando, mas diferentes op codes.

Esta relação ilustra uma importante diferença entre assembly e linguagem de máquina. Ao observar um programa escrito em linguagem assembly 6502B/65C02, o mo-

do de endereçamento usado em cada comando do programa é determinado pelo operando do comando. Quando um programa 6502B/65C02 é traduzido para linguagem de máquina, o modo de endereçamento usado em cada comando pode ser determinado observando-se o op code do comando.

Veja, agora, a Tabela 7-2, que ilustra os modos de endereçamento do 65C02 e do 6502B.

Tabela 7-1 Diferenças nos Modos de Endereçamento em Linguagem Assembly e Linguagem de Máquina

COLUNA 1 MODO DE ENDEREÇAMENTO	COLUNA 2 COMANDOS EM LINGUAGEM ASSEMBLY	COLUNA 3 EQUIVALENTES EM LINGUAGEM DE MÁQUINA	COLUNA 4 NÚMERO DE BYTES
Imediato	ADC#$03	69 03	2
Página Zero	ADC#$03	65 03	2
Página Zero, X	ADC $03,X	75 03	2
Absoluto	ADC $0300	6D 00 03	3
Absoluto Indexado, X	ADC $0300,X	7D 00 03	3
Absoluto Indexado, Y	ADC $0300,Y	79 00 03	3
Indexado Indireto	ADC ($03,X)	61 03	2
Indireto Indexado	ADC ($03),Y	71 03	2
Página Zero Indireto*	ADC ($03)	72 03	2

* Somente 65C02

Tabela 7-2 Modos de Endereçamento do Microprocessador 65C02/6502B

MODO DE ENDEREÇAMENTO	FORMATO
1. Implícito (Implicado)	RTS
2. Acumulador	ASL A
3. Imediato	LDA # 2
4. Absoluto	LDA $ 02A7
5. Página Zero	STA $33
6. Relativo	BCC LABEL
7. Absoluto Indexado, X	LDA $9000,X
8. Absoluto Indexado, Y	LDA $9000,Y
9. Página Zero, X	LDA $33,X
10. Página Zero, Y	STX $33,Y
11. Indexado Indireto	LDA ($33,X)
12. Indireto Indexado	LDA ($33),Y
13. Absoluto Indireto	JMP ($089A)
14. Página Zero Indireto*	ADC ($0A)
15. Absoluto Indexado Indireto*	JMP ($089A,X)

* Somente 65C02

Dos quinze modos ilustrados, somente dois foram até agora utilizados neste livro: o modo *imediato* (como LDA #2) e o modo *absoluto* (como LDA $0207). Como vimos no Capítulo 2, o operando de um comando escrito no modo de endereçamento imediato é sempre um valor numérico, enquanto o operando de um comando escrito em modo absoluto é sempre um endereço de memória. Se o comando em modo imediato LDA #2 fosse encontrado durante a execução de um programa em linguagem assembly, o valor 2 seria carregado no acumulador. Entretanto, se o comando em modo absoluto LDA $0207 fosse o encontrado, o valor armazenado no registro de memória $0207 seria carregado no acumulador.

OBSERVANDO NOVAMENTE O PROGRAMA ADDNRS

O Programa 7-1 é uma listagem do ADDNRS.SRC, um programa em linguagem assembly que já utilizamos anteriormente, e que foi aqui repetido não somente por conter vários modos de endereçamento, mas, também, por ilustrar como esses modos são utilizados na linguagem assembly 6502B/65C02.

Esta versão do programa foi escrita com o sistema assembler/editor Merlin Pro; entretanto, se você possui um assembler Apple ProDOS ou ORCA/M, deverá já estar apto a alterar o programa de forma a atender aos requisitos de seu sistema sem maiores problemas, uma vez que as principais diferenças entre os formatos foram descritas nos capítulos anteriores.

Programa 7-1
O PROGRAMA ADDNRS, SR2

```
 1 *
 2 * ADDNRS.SR2
 3 *
 4 . ORG $8000
 5 *
 6 ADDNRS CLD
 7 CLC ;        Endereçamento Implícito
 8 LDA#2 ;      Endereçamento Imediato
 9 ADC#3;       Endereçamento Imediato
10 STA $0300 ;Endereçamento Absoluto
11 RTS
```

O programa utiliza três modos de endereçamento, identificados na coluna de comentários. Vamos, agora, examinar cada um desses três modos usados na rotina ADDNRS. (O asterisco indica os modos reconhecidos somente pelo 65C02.)

ENDEREÇAMENTO IMPLÍCITO

O modo de endereçamento implícito é um modo que não requer – e não permite – um operando. Ao usar o endereçamento implícito, tudo o que se tem a fazer é digitar um mnemônico de três letras. Nunca será necessário especificar um operando, pois já está contido em cada mnemônico usado no modo de endereçamento implícito.

Do ponto de vista de gerenciamento da memória, é bom utilizar tantos mnemônicos do modo implícito quantos forem possíveis. Por não utilizarem operandos, eles requerem apenas um byte de memória, contra os dois ou três bytes consumidos por comandos escritos em outros modos de endereçamento.

São exemplos de endereçamento implícito:

```
CLC     ;(DESATIVAR O BIT DE TRANSPORTE DO REGISTRO P)
DEX     ;(DECREMENTAR O REGISTRO X)
INY     ;(INCREMENTAR O REGISTRO Y)
```

Os mnemônicos op-code que podem ser usados no modo de endereçamento implícito são BRK, CLC, CLD, CLL, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PHX*, PHY*, PLA, PLP, PLX*, PLY*, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS e TYA.

ENDEREÇAMENTO IMEDIATO

O modo de endereçamento imediato sempre requer um operando, que é sempre um número. Num comando que usa endereçamento imediato, portanto, sempre aparece um sinal "#" à frente do operando.

Quando um endereço imediato é usado num comando em assembly, o assembler não precisa ir a uma localização de memória para encontrar um valor; este está presente no comando e é colocado diretamente no acumulador. A operação que o comando chama pode ser realizada imediatamente.

Um comando escrito em modo de endereçamento imediato necessita sempre de dois bytes de memória: um para o op code e outro para o operando. Os comandos em modo imediato nunca usam operandos de dois bytes, uma vez que o 6502B/65C02 não manuseia números maiores que os de um byte.

Eis alguns exemplos de endereçamento imediato:

```
LDA     #2
ADC     #$33
SBC     #253
```

As instruções que podem ser utilizadas no modo de endereçamento imediato são ADC, AND, BIT*, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA e SBC.

ENDEREÇAMENTO ABSOLUTO

Em um comando que utiliza o modo de endereçamento absoluto, o operando é um endereço de memória, não um número. O mnemônico, neste caso, sempre pede a realização de uma operação com um valor armazenado num endereço de memória especificado, não com o próprio operando.

O operando, sendo um endereço de memória, sempre requer dois bytes. Como o Apple pode manusear endereços de até 16 bits, dois bytes são sempre reservados para operandos usados no modo de endereçamento absoluto.

Eis alguns exemplos de comandos em linguagem assembly escritos no modo de endereçamento absoluto:

```
LDA     $0300
STA     768
CMP     $FFD0
```

Os mnemônicos que podem ser usados no modo de endereçamento absoluto são ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY, STZ*, TSB* e TRB*.

ENDEREÇAMENTO DA PÁGINA ZERO

O *Endereçamento da Página Zero* é bastante similar ao endereçamento absoluto. Quando um comando é escrito usando um endereço página zero, um valor é retirado de um endereço de memória específico, e este será o valor no qual o operando trabalha.

Há uma importante diferença, entretanto: quando um comando é escrito em endereçamento absoluto, o endereço acessado pode pertencer a qualquer região da RAM disponível. Mas, no endereçamento página zero, o endereço deve, necessariamente, estar

compreendido em uma área especial da memória, chamada Página Zero.

Especificamente, o bloco de memória que o computador conhece como Página Zero ocupa os 256 registros de memória que se estendem do endereço $00 até o $FF. Pode-se, também, dizer que a Página Zero estende-se de $0000 a $00FF, mas não é realmente necessário usar os dois zeros extras, ao referir-se a um endereço da Página Zero. Usualmente, quando uma instrução em assembly é seguida por um endereço de um byte – ou por um número hexa de quatro dígitos que começa com dois zeros – o assembler, automaticamente, interpretará o endereço especificado como sendo um endereço da Página Zero.

Como só é necessário um byte para especificar um endereço da Página Zero, o comando requererá somente dois bytes de memória, um para o op code e outro para o operando. Geralmente, porém, não há muito espaço livre de memória na Página Zero; usualmente, não é possível utilizar grande quantidade de endereçamento da Página Zero nos programas em assembly.

No Capítulo 11, dedicado ao gerenciamento da memória, haverá mais informações sobre o espaço da Página Zero e sua utilização. Por ora, seguem alguns exemplos que mostram com o que se parece o endereçamento da Página Zero nos programas em linguagem assembly.

Se seu computador é equipado com um microprocessador 6502B, as instruções que podem ser usadas no modo de endereçamento da Página Zero são ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX e STY. Se seu Apple tem um 65C02, você pode também utilizar o endereçamento da Página Zero para diversas instruções novas: STZ (store a zero in memory = armazenar um zero na memória), TRB (test and reset memory bits with accumulator = testar e desativar bits da memória com os do acumulador) e TSB (test and set memory bits with accumulator = testar e ativar bits da memória com os do acumulador).

ENDEREÇAMENTO DO ACUMULADOR

Este modo pode ser usado com vários mnemônicos que realizam operações com o valor armazenado no acumulador do 6502B/65C02. O comando ASL A, por exemplo, é usado para deslocar cada bit do acumulador uma posição, com um 0 tomando o lugar do bit mais à direita, e com o bit mais à esquerda (bit 7), indo para o bit de transporte (carry bit) do registro de status do processador (P).

Se seu computador é equipado com o 6502B, outras instruções que podem ser usadas no modo de endereçamento do acumulador são LSR, ROL e ROR. Se é equipado com o 65C02, podem ser utilizados também os mnemônicos INA ou INC A (incrementar acumulador) e DEA (decrementar acumulador).

Caso seu assembler seja o ORCA/M, você deverá usar a letra A como operando, ao utilizar o modo de endereçamento do acumulador. O Merlin Pro e o Apple ProDOS Assembler, entretanto, aceitarão mnemônicos de endereçamento do acumulador sem o operando A.

ENDEREÇAMENTO RELATIVO

O *endereçamento relativo* é usado com uma técnica de programação denominada *desvio condicional* (conditional branching), cujas instruções são similares às instruções IF...GO-TO em BASIC; elas são utilizadas para fazer com que o programa salte uma seqüência de instruções quando determinadas condições forem satisfeitas.

Existem oito instruções de desvio condicional que podem ser usadas com o 6502B, e o 65C02 aceita ainda mais uma. Essas nove instruções começam pela letra B, que significa *branch to*. A instrução extra que pode ser utilizada com o 65C02 é BRA, que significa *branch always* (desviar sempre). Como o nome indica, o mnemônico BRA causará um desvio para outro segmento do programa em qualquer condição.

As outras oito instruções de desvio condicional que usam endereçamento relativo são:

BCC (branch to a specified address if the carry flag is clear = desviar para um endereço especificado se o indicador de transporte estiver desativado)

BCS (branch if the carry flag is set = desviar se o indicador de transporte estiver ativado)

BEQ (branch if the result of an operation is equal to 0 = desviar se o resultado de uma operação for igual a 0)

BNE (branch is the result of an operation is not equal to 0 = desviar se o resultado de uma operação for diferente de 0)

BMI (branch if the result of an operation is minus = desviar se o resultado de uma operação for negativo)

BPL (branch if the result of an operation is plus = desviar se o resultado de uma operação for positivo)

BVC (branch if the overflow flag is clear = desviar se o indicador de "overflow" estiver desativado)

BVS (branch if the overflow flag is set = desviar se o indicador de "overflow" estiver ativado).

Como são usadas as Instruções de Desvio As nove instruções de desvio condicional da linguagem assembly 6502B/65C02 são, geralmente, usadas em conjunto com outras instruções, as *instruções de comparação*. Estas, tipicamente, são usadas para comparar dois valores entre si, e a instrução de desvio é então utilizada para determinar o que deve ser feito, dependendo do resultado da comparação.

Normalmente, uma instrução de desvio faz com que o programa salte para um endereço especificado, caso determinadas condições sejam ou não satisfeitas. Um desvio pode ocorrer, por exemplo, se um número for maior que outro, se dois números forem iguais, ou se o resultado de determinada operação for positivo, negativo ou igual a zero.

Os desvios condicionais podem, também, ser baseados nos resultados de operações aritméticas ou lógicas, ou chamados após vários tipos de verificações em bits, bytes ou valores numéricos.

As três instruções de comparação na linguagem assembly 6502B/65C02 são:

- CMP (compare o número no acumulador com ...)
- CPX (compare o valor do registro X com ...)
- CPY (compare o valor do registro Y com ...)

Um Exemplo de Desvio Condicional O Programa 7-2, intitulado ADDCHK, é um exemplo de rotina em linguagem de máquina utilizando desvio condicional. (A rotina foi digitada com o Merlin Pro, mas pode ser facilmente adaptada para o Apple ProDOS ou para o ORCA/M.)

Program 7-2
ADDCHK
Um Programa de Edição com Verificação de Erro

```
 1 *
 2 * ADDCHK
 3 *
 4  ORG $8000
 5 *
 6 ADDCHK LDA #0
 7  STA $9000
 8 *
 9  CLD
10  CLC
11  LDA $0300
12  ADC $0301
13  BCS ERROR
14  RTS
15 *
16 ERROR LDA #1
17  STA $9000
18  RTS
```


O ADDCHK é uma rotina de adição de 8 bits com um simples utilitário de verificação de erros incorporado. O programa adiciona dois valores de 8 bits, usando endereçamento absoluto. Se o resultado do cálculo for um valor de 16 bits (um número maior que 255), haverá um erro de transbordamento e o bit de transporte (carry bit) do registro de status do processador será ativado.

Eis como o ADDCHK funciona: na seqüência de duas linhas, intitulada ADDCHK, um 0 é carregado no registro de memória $9000. A seguir, os indicadores de transporte e de decimal são desativados, e o valor do registro de memória $0300 é somado com o valor do registro de memória $0301. Se esta adição resulta num transporte, o bit de transporte do registro de status do processador será ativado automaticamente.

Após a operação de adição, uma instrução BCS (desviar se o transporte estiver ativado) é usado para verificar o bit de transporte do registro P. Se o bit de transporte estiver ativado, o programa pula para uma rotina denominada ERROR. Se o bit de transporte não estiver ativado, o programa termina.

Na rotina ERROR, um indicador – o número 1 – é carregado no registro de memória $9000, e então o programa termina.

ENDEREÇAMENTO ABSOLUTO INDEXADO

Um endereço indexado, da mesma forma que um endereço relativo, é calculado pelo uso de um deslocamento (*offset*). Num endereço indexado, entretanto, o deslocamento é determinado pelo conteúdo atual dos registros X ou Y do 6502B/65C02.

Um comando contendo um endereço indexado pode ser escrito num destes formatos:

```
LDA $02A7, X
LDA $02A7, Y
```

Como Funciona o Endereçamento Absoluto Indexado Quando o endereçamento indexado é usado num comando em linguagem assembly, o conteúdo do registro X ou do registro Y (dependendo do registro-índice que está sendo utilizado) é adicionado ao endereço fornecido na instrução para determinação do endereço final.

O Programa 7-3, PRINTIT, é um exemplo de programa que utiliza endereçamento indexado. O programa movimenta-se, byte por byte, ao longo de uma seqüência (string) de caracteres ASCII, utilizando uma rotina residente no Apple, chamada COUT, para apresentar cada caractere da seqüência na tela. Quando toda a seqüência houver sido apresentada, o programa termina.

O endereço de início da COUT e o código ASCII para o <return> são definidos na tabela de símbolos que precede o programa.

Maiores detalhes a respeito do funcionamento de programas deste tipo serão fornecidos no Capítulo 8. A finalidade básica deste exemplo é ilustrar o endereçamento indexado.

Programa 7-3
PRINTIT
Um Programa de Apresentação de Caracteres na Tela

```
 1 *
 2 * PRINTIT
 3 *
 4 COUT EQU $FDED
 5 EOL EQU 13
 6 *
 7  ORG $8000
 8 *
 9  JMP PRINTIT
10 *
11 TEXT DB 198,204,193,211,200,160,173,173,160,193,208,208,
   204,197,160
12  DB 207,215,206,197,210,160,194,210,197,193,203,211,160
13  DB 205,193,195,200,201,206,197,160,195,207,196,197,
   161,13
14 *
15 PRINTIT  LDX  #0
16 LOOP LDA TEXT,X
17  JSR COUT
18  CMP #EOL
19  BEQ FINI
20  INX
21  JMP LOOP
22 FINI RTS
23  END
```


Procurando por um <Return> Quando o Programa 7-3 inicia, nós sabemos que a seqüência termina com um <Return> (código ASCII $0D), como em geral o fazem as seqüências em programas Apple.

Conforme o programa prossegue trabalhando na seqüência, ele verifica cada caractere. Se o caractere não é um <Return>, o programa segue para o próximo caractere. Se o caractere for um <Return>, isto significa o término da seqüência, e a rotina termina.

Endereçamento absoluto indexado pode ser usado com as seguintes instruções 6502B/65C02: ADC, AND, ASL (somente X), CMP, DEC (somente X), EOR, INC (somente X), LDA, LDX (somente Y), LDY (somente X), LSR (somente X), ORA, ROL (somente X), ROR (somente X), SBC e STA. Caso seu computador possua o 65C02, você poderá utilizar dois mnemônicos adicionais – BIT e STZ (armazenar zero) – no modo de endereçamento absoluto indexado (somente X).

ENDEREÇAMENTO DA PÁGINA ZERO, X

O Endereçamento da Página Zero, X é utilizado exatamente como o Absoluto Indexado, X. Porém, os endereços utilizados precisam pertencer à Página Zero. Assim, esta forma de endereçamento usa somente um byte de memória como operando, quando convertido em linguagem de máquina.

As instruções que podem ser utilizadas no modo de endereçamento da Página Zero, X são ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA e STY. Caso seu computador esteja equipado com o 65C02, você poderá também utilizar os mnemônicos BIT e STZ (armazenar zero).

ENDEREÇAMENTO DA PÁGINA ZERO, Y

A utilização é idêntica à do endereçamento da Página Zero, X; porém, somente podem ser usados dois mnemônicos: LDX e STX. Se não fosse pelo modo de endereçamento da Página Zero, Y, seria impossível usar o modo de endereçamento absoluto indexado com as instruções LDX e STX.

ENDEREÇAMENTO INDIRETO

O *Endereçamento Indireto* pode ser dividido em duas subcategorias: *endereçamento indexado indireto* e *endereçamento indireto indexado*. Estes nomes podem parecer confusos, mas há um truque para diferenciá-los: Indexado indireto, que tem um X na primeira palavra, é um modo de endereçamento que faz uso do registro X do 6502B/65C02. Indireto indexado, que ***não*** tem um X na primeira palavra, usa o registro Y.

Ambos os modos são basicamente utilizados para procurar dados armazenados em tabelas.

Endereçamento Indexado Indireto. Três fatos acontecem quando um programa usa endereçamento indexado indireto. Primeiro, o conteúdo do registro X é adicionado a um endereço da Página Zero – não ao conteúdo do endereço, mas ao próprio endereço. A seguir, o resultado deste cálculo é interpretado como outro endereço da Página Zero. Finalmente, após o cálculo deste segundo endereço, o valor que ele contém e o conteúdo do próximo endereço são combinados para obtenção de um terceiro endereço. Este terceiro endereço é, finalmente, interpretado como o operando do comando em questão.

Vejamos um exemplo que deverá esclarecer este processo: suponha que o endereço de memória $A0 de seu computador contenha o número $00, que o endereço $A1 contenha o número $80 e que o registro X contenha o número 00.

Eis uma tabela ilustrativa desses valores:

```
$A0 = #$00
$A1 = #$80
  X = #$00
```

Vamos, agora, supor que esteja sendo rodado um programa que inclua a instrução indexada indireta LDA ($A0,X).

Ao encontrar essa instrução, o computador adicionará o conteúdo do registro X (um 0) ao número $A0. A soma de $A0 e 0 é, logicamente, $A0. O computador irá, então, aos endereços de memória $A0 e $A1, e encontrará o número $00 no endereço $A0 e o número $80 no endereço $A1.

Uma vez que os computadores baseados no 6502B/65C02 armazenam números de 16 bits em ordem inversa – primeiramente o byte inferior – seu computador interpretará o número encontrado em $A0 e $A1 como $8000. E irá, então, carregar o acumulador com o valor que encontrar armazenado no endereço $8000.

Imagine agora que, quando o computador encontra o comando LDA ($A0, X), o registro X do processador contenha o número 04, em lugar de 00.

Eis uma tabela ilustrando aqueles valores e mais alguns equivalentes que utilizaremos brevemente:

```
$A0 = #$00
$A1 = #$80
$A2 = #$0D
$A3 = #$FF
$A4 = #$FC
$A5 = #$1C
  X = #$04
```

Se estas condições existirem quando o computador encontrar a instrução LDA ($A0, X), ele somará o número $04 (o valor do registro X) ao número $A0, e irá aos endereços $A4 e $A5. Nestes dois endereços, encontrará o endereço final (byte inferior em primeiro lugar, é claro) dos dados que está procurando – neste caso – $1CFC.

Endereçamento indexado indireto é um modo de endereçamento não muito usado em programas em linguagem assembly. Quando usado, sua finalidade é a de localizar um endereço de 16 bits numa tabela de endereços armazenada na Página Zero.

Espaço na Página Zero, entretanto, é difícil de se encontrar; você não poderá, provavelmente, armazenar ali muitas tabelas de endereços. Portanto, não é fácil encontrar muita utilidade para este modo de endereçamento.

Endereçamento Indireto Indexado Este modo é usado com mais freqüência que o modo indexado indireto, em programas em linguagem assembly 6502B/65C02.

O endereçamento indireto indexado usa o registro Y (nunca o registro X) como um *offset* para calcular o endereço de início de uma tabela. O endereço de início (ou endereço-base) da tabela tem de ser armazenado na Página Zero, mas não a tabela propriamente dita.

Quando um assembler encontra um endereço indireto indexado em um programa, ele primeiramente procura pelo endereço da Página Zero que está entre os parênteses que precedem o Y. O valor de 16 bits armazenado nesse endereço e o endereço seguinte é, então, somado ao conteúdo do registro Y. O valor resultante é um endereço de 16 bits – o endereço que o comando estava procurando.

Vejamos um exemplo de endereçamento indireto indexado: se seu computador, rodando um programa, chega a uma instrução ADC ($A0), Y, então verificará os endere-

ços de memória $A0 e $A1. Suponha que ele encontre o número $00 no registro $A0 e o número $50 no registro $A1. Suponha, também, que o registro Y contém um 4.

Eis a lista destas condições:

```
$A0 = #$00
$A1 = #$50
 Y  = #$04
```

O computador combinará os números $00 e $50 (byte inferior primeiro) e encontrará o endereço $5000. Adicionará, então, o conteúdo do registro Y (neste caso, 4) e chegará ao total de $5004.

O número $5004 é o valor final do operando ($A0), Y. Assim, o conteúdo do acumulador será adicionado ao número que estiver armazenado do endereço de memória $5004.

Compreendido o endereçamento indireto indexado, ele pode-se tornar uma valiosa ferramenta na programação em assembly. Somente um endereço – o endereço de início de uma tabela – tem de ser armazenado na Página Zero, onde o espaço é sempre reduzido. Já esse endereço, somado ao conteúdo do registro Y, pode ser usado como um ponteiro, para localizar qualquer outro endereço na memória do computador.

Endereçamento Absoluto Indireto Existe uma única instrução – JMP – que pode ser utilizada no modo de *endereçamento absoluto indireto*. Quando este modo é utilizado, um número de 16 bits é colocado entre os parênteses que seguem a instrução JMP, como

```
JMP ($0900)
```

Este número atua como um ponteiro para dois registros de memória que, juntos, contêm o endereço para o qual o programa deve pular. Suponhamos, por exemplo, que o endereço de memória $02A7 contenha o valor $00, e que o endereço $02A8 contenha o valor $06. Vamos supor, também, que o comando JMP ($02A7) esteja incluído em um programa Apple em assembly. Neste caso, o programa em execução pulará ao endereço $0600 – *não* ao endereço $02A7, como o faria se a instrução fosse simplesmente JMP $02A7.

Endereçamento Absoluto Indexado Indireto Este modo também só pode ser utilizado com a instrução JMP, e somente nos computadores equipados com o 65C02. Este é o formato para a escrita de um comando no endereçamento absoluto indexado indireto:

```
JMP ($8000, X)
```

Quando este modo é utilizado num programa, o conteúdo do endereço que segue a instrução é adicionado ao registro X. O resultado desta operação de adição aponta para um endereço de memória que contém o byte inferior do endereço efetivo. A localização de memória seguinte contém o byte superior do endereço efetivo.

Endereçamento da Página Zero, Indireto. Este é um novo modo de endereçamento, somente utilizável em computadores equipados com o microprocessador 65C02. Este é o formato para o modo Página Zero indireto:

```
LDA ($0A)
```

Quando este modo é utilizado em um programa, o byte que segue a instrução é interpretado como um endereço da Página Zero. Este byte contém o byte de ordem inferior do endereço efetivo, e o próximo endereço da Página Zero contém o byte de ordem superior.

Este modo pode ser utilizado com as seguintes instruções: ADC, AND, CMP, EOR, LDA, ORA, SBC e STA.

UM PSEUDO-ENDEREÇO: A PILHA

Enquanto tratamos dos modos de endereçamento do 6502B/65C02, vamos dar uma olhada numa ferramenta de programação que é estreitamente relacionada ao endereçamento: a *pilha de hardware* (hardware stack).

A pilha de hardware – ou, simplesmente, pilha – ocupa 256 bytes de memória, de $0100 a $01FF, em RAM.

A pilha é o que os programadores algumas vezes chamam bloco de memória LIFO (last-in, first-out = entra por último, sai primeiro), e pode ser comparada a um dispositivo de guardar moedas onde elas fazem pressão contra uma mola: cada moeda que é inserida empurra as outras, ficando por cima. A última moeda inserida será a primeira a ser retirada. Quando se coloca um número na localização de memória que é o topo da pilha, ele cobre o número que ali se encontrava previamente. O número colocado no topo tem de ser agora removido, antes que se possa acessar o número que ficou sob ele.

Embora a comparação com o moedeiro possa servir para descrever o funcionamento da pilha, ela não é exata; na realidade, a pilha não é mais que um bloco de RAM. Vejamos o que acontece quando colocamos um número na pilha de hardware do Apple.

O bloco de memória que constitui a pilha (do registro de memória $0100 ao $01FF) é usado de cima para baixo: o primeiro número armazenado ocupará o registro $01FF, o próximo ocupará o registro $01FE, e assim por diante. Devido a este sistema de armazenamento, o último endereço da pilha que pode ser usado é o registro $0100.

O 6502B ou 65C02 do Apple mantém o controle da manipulação da pilha com a ajuda de um registro especial, chamado *ponteiro da pilha* (stack pointer), brevemente descrito no Capítulo 3. Quando não há nada armazenado na pilha, o valor do ponteiro é $FF. Se adicionarmos $100 a esse número, obteremos $01FF – o endereço mais alto da pilha. Este é o endereço que será usado para o próximo (neste caso, o primeiro) valor armazenado na pilha.

Assim que um valor é armazenado na pilha, o processador automaticamente decrementa o ponteiro em uma unidade. A cada armazenamento, o ponteiro é novamente decrementado. Assim, ele apontará sempre para o endereço do próximo valor a ser armazenado na pilha.

Vamos supor que vários números tenham sido armazenados na pilha, e que agora desejamos recuperar um deles. Quando um número que havia sido armazenado na pilha é trazido de volta, o valor do ponteiro da pilha é incrementado unitariamente. Isto efetivamente remove um valor da pilha, uma vez que significa que o próximo valor armazenado na pilha terá a mesma posição que o valor removido.

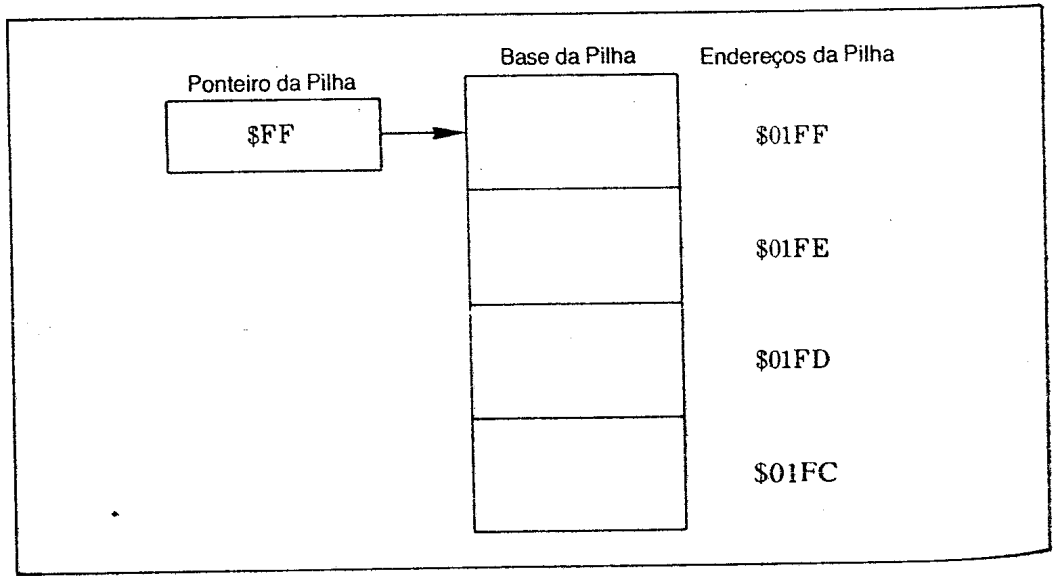


Figura 7.1 Como funciona o ponteiro da pilha.

Examinando a Figura 7-1, você terá uma idéia do funcionamento deste processo. A figura mostra uma pilha vazia, com o ponteiro indicando o primeiro endereço disponível na pilha: $01FF.

Na Figura 7-2, um número (cujo valor é arbitrário) foi colocado na pilha. Note que o valor do ponteiro foi decrementado. O número colocado na pilha está armazenado no endereço mais alto daquela, o registro de memória $01FF.

Na Figura 7-3, outro número (também de valor arbitrário) foi colocado na pilha. Como se pode notar, o ponteiro foi novamente decrementado, e o segundo número está agora na pilha.

Agora, removeremos um número da pilha (Figura 7-4). O endereço da pilha $01FE ainda contém o valor $33, mas o valor do ponteiro da pilha foi incrementado e agora aponta o endereço de memória $01FE. O próximo número que for colocado na pilha, portanto, será armazenado no endereço de memória $01FE. Quando esse número for armazenado, o número armazenado previamente – $33 – será apagado.

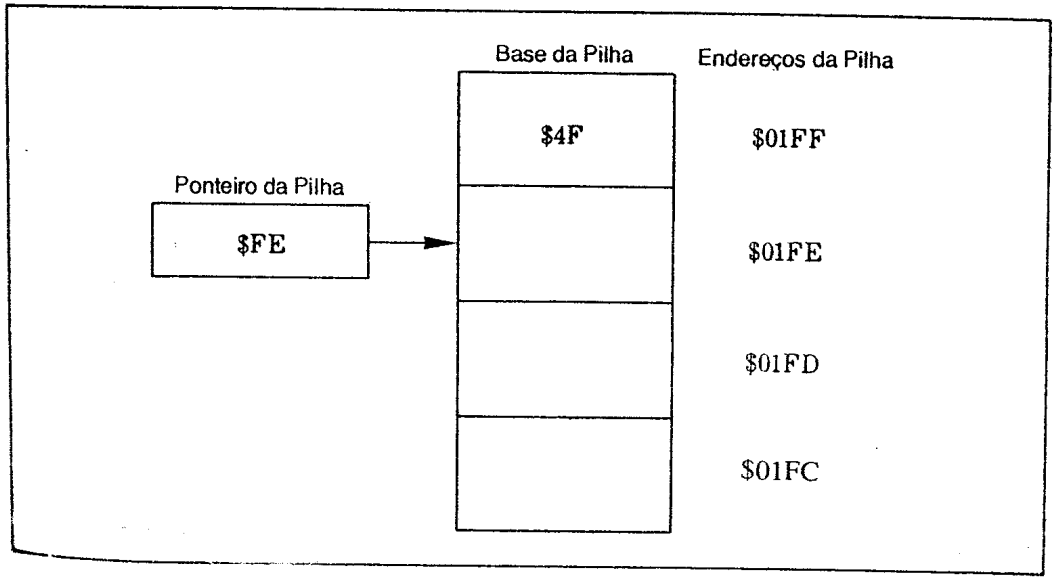


Figura 7.2 Colocando um número na pilha.

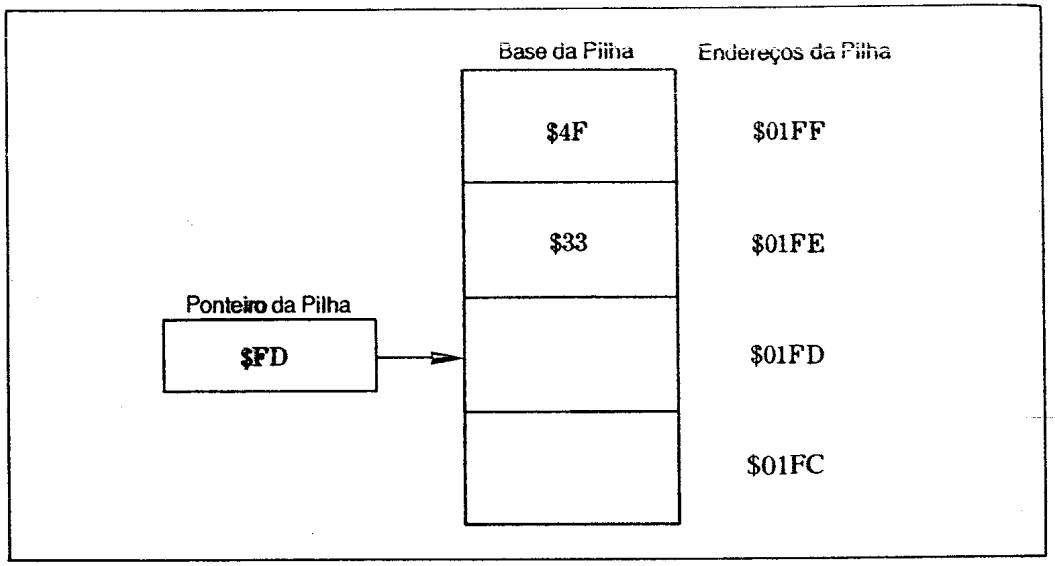


Figura 7-3 Colocando outro número na pilha.

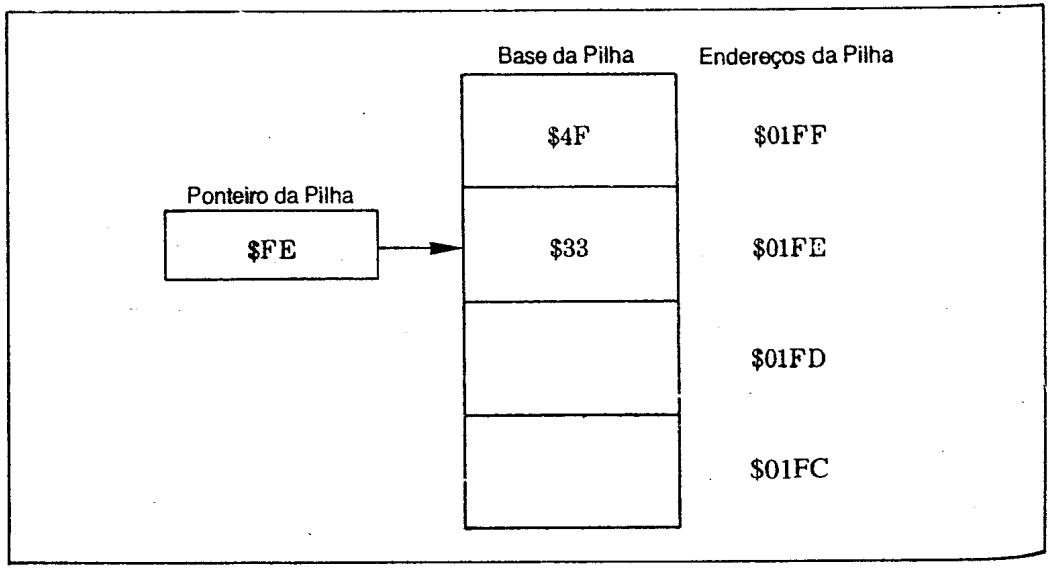


Figura 7-4 Retirando um número da pilha.

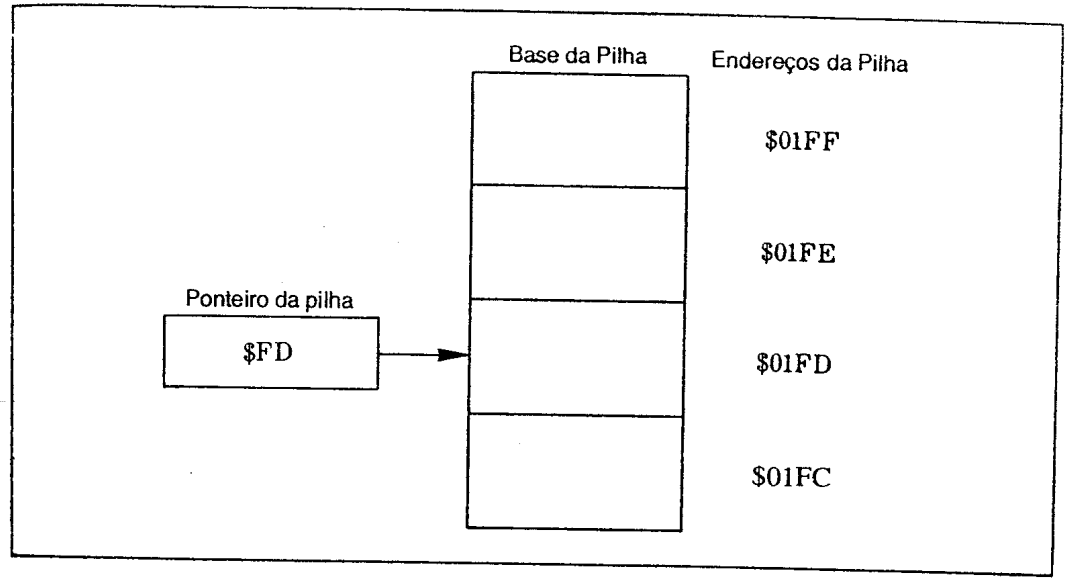


Figura 7-5 Uma última manipulação da pilha.

Vamos, então, armazenar mais um número na pilha. O valor será $17, como mostra a Figura 7-5. O registro de memória $01FE agora contém o valor $17. O valor do ponteiro foi decrementado, o valor $33 apagado pelo valor $17, e o próximo número que for colocado na pilha será armazenado no registro de memória $01FD.

COMO O SISTEMA OPERACIONAL USA A PILHA

O processador 6502B/65C02, gealmente, usa a pilha para armazenamento temporário de dados, durante a operação de um programa. Quando um programa pula para uma sub-rotina, por exemplo, o 6502B/65C02 coloca no topo da pilha o endereço para o qual o programa terá de retornar.

Assim, quando a sub-rotina termina, com a instrução RTS, o endereço de retorno é retirado do topo da pilha e carregado no contador de programa do 6502B/65C02. O programa, então, retorna ao endereço adequado, e o processamento prossegue normalmente.

A pilha também é bastante utilizada em programas escritos pelo usuário. A seguir, apresentaremos um exemplo de rotina que faz uso da pilha. Poderemos reconhecer essa rotina como uma variação do programa de adição com 8 bits que usamos anteriormente.

UM PROGRAMA EM DUAS PARTES

No Programa 7-4, colocaremos dois números de 8 bits na pilha. No Programa 7-5, nós os retiraremos da pilha e efetuaremos uma adição. O Programa 7-4 deve ser digitado em primeiro lugar; porém, antes de ser compilado e executado, o Programa 7-5 deve ser anexado (appended) a ele.

Programa 7-4
STACKADD, PARTE 1
(Colocando dois números na pilha)

```
1  *
2  *STACKADD
3  *
4    ORG $8000
5  *
6    LDA #35  ; (OU QUALQUER OUTRO NUMERO DE 8 BITS)
7    PHA
8    LDA #49  ; (OU QUALQUER OUTRO NUMERO DE 8 BITS)
9    PHA
```

Anexemos agora o Programa 7-5 ao final do 7-4.

Programa 7-5
STACKADD, PARTE 2
(Retirando dois números da pilha e somando os mesmos)

```
 1  *
 2  *QUANDO ESTE PROGRAMA INICIA,
 3  *DOIS NUMEROS ESTAO NA PILHA
 4    CLD
 5    CLC
 6    PLA
 7    STA $FD
 8    PLA
 9    ADC $FD
10    STA $FE
11    RTS
```

Anexando os dois programas obteremos uma rotina de adição em 8 bits, que demonstra a conveniência do uso da pilha em programas em assembly. Primeiramente, um valor é retirado da pilha e armazenado no acumulador e, também, no endereço de memória $FD.

A seguir, outro valor é retirado da pilha e somado ao valor agora armazenado em $FD. O resultado deste cálculo é, então, armazenado em $FE e a rotina termina.

Como se pode ver, a pilha pode ser uma área muito conveniente para armazenamento temporário de dados. A pilha é, também, bastante eficiente em termos de memória, uma vez que não requer o uso de registros de armazenamento dedicados. E pode também economizar tempo, pois necessitamos apenas de uma instrução para colocar um valor na pilha e de uma instrução para retirá-la.

UM AVISO IMPORTANTE

Deve-se ter muito cuidado ao utilizar a pilha em uma rotina em assembly. Quando a rotina termina, é muito importante deixá-la exatamente como foi encontrada. Se foi colocado um valor na pilha, durante o curso de uma rotina, ele deve ser removido antes que a rotina termine e o processamento normal continue.

A não-remoção do valor pode resultar em "travamento" do programa, apagamento de memória ou outros desastres de programação. Porém, se manuseada adequadamente, a pilha torna-se uma poderosa ferramenta de programação.

Os mnemônicos que fazem uso da pilha são PHA (push the contents of the accumulator onto the stack = empurrar o conteúdo do acumulador para a pilha), PLA(pull the top value off the stack and deposit in the accumulator = retirar o valor do topo da pilha e depositá-lo no acumulador), PHP (push the contents of the P register onto the stack = empurrar o conteúdo do registro P para a pilha) e PLP (pull the top value off the stack and deposit it into the P register = retirar o valor do topo da pilha e depositá-lo no registro P). As instruções JSR e RTS do 65C02 PHX, PHY, PLX e PLY também fazem uso da pilha.

As operações PHP e PLP são, geralmente, incluídas nas sub-rotinas em assembly, de forma a evitar que se apague o conteúdo do registro P durante as sub-rotinas. Quando "pulamos" para uma sub-rotina que altera o estado do registro P, devemos iniciar a sub-rotina colocando o conteúdo do registro P na pilha. Assim, imediatamente antes do final da sub-rotina, podemos restaurar o estado anterior do registro P, com uma instrução PHP.

CAPÍTULO 8

"LOOPS" E DESVIOS

Agora que você já está familiarizado com o jogo de instruções e os modos de endereçamento do 6502B/65C02, podemos começar realmente a programar em linguagem assembly Apple IIc/IIe.

Neste capítulo, você aprenderá a apresentar mensagens na tela, codificar e decodificar caracteres ASCII e realizar diversos outros procedimentos úteis em linguagem assembly, como:

- Incrementar e decrementar os registros X e Y.
- Usar, em conjunto, instruções de comparação e desvio.
- *Loops* e desvios.
- Usar diretivas Apple assembler: DFB ou DB (definir byte) e DC (definir constante, uma diretiva usada com o assembler ORCA/M), que são usadas para reservar espaço na memória para dados em programas em linguagem assembly.

Neste capítulo, utilizaremos novamente a rotina COUT, residente nos Apple IIc e IIe, iniciando no endereço $FDED. Nós a utilizamos no Capítulo 1, para apresentar a palavra "HI" na tela, no Programa HI.TEST.

UM NOVO PROGRAMA

O Programa 8-1, BULLETIN, é o mais extenso programa do tipo "digite e rode" até aqui apresentado neste livro. Foi escrito com o Merlin Pro, mas rodará também com o Apple ProDOS. A versão ORCA/M do programa é apresentado no Programa 8-2.

Programa 8-1
O PROGRAMA BULLETIN
(Versão Merlin Pro)

```
 1 *
 2 * BULLETIN
 3 *
 4  ORG $8000
 5 *
 6 BUFLEN EQU 29
 7 COUT EQU $FDED
 8 *
 9  JMP BEGIN
10 TEXT DB 195,197,204,197,211,212,201,193,204,160
11  DB 193,195,195,201,196,197,206,212,160
12  DB 210,197,208,207,210,212,197,196,173,173
13 *
14 BEGIN LDX #0
15 *
16 LOOP LDA TEXT,X
17  JSR COUT
18  INX
19  CPX #BUFLEN
20  BNE LOOP
21  RTS
```

Após digitar, compilar e executar o programa BULLETIN, você verá metade de uma mensagem enigmática na tela. A outra metade será apresentada mais adiante, neste capítulo, no programa BULLETIN.B.

GRAVANDO O PROGRAMA BULLETIN

Uma vez que o programa BULLETIN esteja rodando corretamente, grave-o em disco em ambas as versões, código-fonte e código-objeto. A seguir, vamos ver como o programa funciona.

Comecemos com uma explanação sobre a diretiva DB, que significa "definir byte". Ela aparece nas linhas 10, 11 e 12 da versão Merlin/Apple do programa. (Na listagem ORCA/M, é usada uma diretiva ligeiramente diferente: DC, que significa "definir constante". Esta diretiva será discutida logo adiante.)

Diretivas como DB e DC são pseudocódigos operacionais (pseudo-ops); aparecem na coluna op code da listagem em código-fonte do programa em assembly, mas, na realidade, não fazem parte do jogo de instruções-padrão em linguagem assembly

6502B/65C02. Como mencionamos no Capítulo 4, a principal diferença entre um op code e um pseudo-op é que o primeiro diz ao processador o que este deve fazer, enquanto o pseudo-op diz ao assembler o que fazer.

Quando um programa roda através de um assembler, cada op code que o programa contém é traduzido em linguagem de máquina, enquanto cada pseudo-op encontrado é usado como uma instrução pelo assembler.

Ao encontrar a diretiva DB, por exemplo, o assembler interpreta literalmente os números que seguem a diretiva, depositando-os em uma série de registros consecutivos na memória do computador. Você conhecerá vários outros tipos de pseudo-ops nos capítulos subseqüentes.

Vejamos, agora, a diretiva DC, usada em lugar da DB na versão ORCA/M do programa BULLETIN.

Programa 8-2
O PROGRAMA BULLETIN
(Versão ORCA/M)

```
          KEEP   BULLETIN
MAIN      START

BUFLEN    EQU    29
COUT      EQU    $FDED

          JMP    BEGIN

TEXT      DC     I1'195,197,204,197,211,212,201,193,204,160'
          DC     I1'193,195,195,201,196,197,206,212,160'
          DC     I1'210,197,208,207,210,212,197,196,173,173'

BEGIN     LDX    #0

LOOP      LDA    TEXT,X
          JSR    COUT
          INX
          CPX    #BUFLEN
          BNE    LOOP
          RTS

          END
```

À diretiva DC, diferentemente da DB, é necessário dizer como interpretar a seqüência de valores que vem a seguir. Na listagem ORCA/M do programa BULLETIN, a letra I, que

segue cada diretiva DC, significa "inteiro", e o número 1, após cada I, indica que cada valor deve ser interpretado como um inteiro de um byte.

USANDO OS REGISTROS X E Y COMO CONTADORES

Como indicado no Capítulo 3, os registros X e Y do 6502B/65C02 podem ser progressivamente incrementados e decrementados durante *loops* nos programas.

No programa BULLETIN, o registro X é incrementado de 0 a 29 durante um *loop* para acompanhamento da seqüência de caracteres que está sendo apresentada na tela. Os caracteres são expressos em código ASCII: os números que seguem as diretivas DB e DC no programa BULLETIN.

Não é difícil perceber como funciona, neste programa, o *loop* para o registro X. Primeiramente, é utilizado o comando LDX#0 para carregar o registro X com um 0. A seguir, inicia o *loop*.

O primeiro comando daquele é "LDA TEXT, X". Esta instrução usa endereçamento indexado para carregar o acumulador com o código ASCII de um caractere de texto. A rotina COUT é, então, utilizada para exibir cada caractere na tela.

Ao término do *loop*, todos os 29 caracteres subseqüentes à diretiva dB ou DC terão sido apresentados na tela.

(Incidentalmente, não há necessidade do símbolo "#" à frente dos números que seguem as diretivas DB e DC, pois eles serão automaticamente interpretados como números pelos assemblers.)

INCREMENTANDO E DECREMENTANDO OS REGISTROS X E Y

Na linha 18 do programa BULLETIN, temos o mnemônico INX, que significa "incrementar o registro X".

Na primeira passagem do programa através do *loop* que começa na linha 16, o registro X contém um 0. Assim que a rotina COUT apresenta o caractere na tela, a instrução INX da linha 18 incrementará o registro X em uma unidade.

Na linha 19, temos a instrução CPX#BUFLEN. Observando a linha 5, verá que BUFLEN é uma constante que foi igualada ao número 29. A instrução CPX #BUFLEN,

portanto, significa "comparar o valor do registro X com o número 29".

Esta comparação verifica se os 29 caracteres já foram apresentados na tela. Uma vez apresentados, pressione <return> para terminar o programa.

COMPARANDO VALORES EM LINGUAGEM ASSEMBLY

Existem três instruções de comparação na linguagem assembly 6502B/65C02: CMP, CPX e CPY.

CMP significa "comparar com o valor do acumulador". Quando a instrução CMP é usada, seguida de um operando, o valor expresso pelo operando é subtraído do valor do acumulador. Esta operação é efetuada não para determinar a diferença exata entre os dois valores, mas somente para verificar se são iguais e, caso não seja, qual é o maior.

Se o valor do acumulador for igual ao valor testado, o indicador de zero (Z) do registro de status do processador (P) será ativado para 1. Se o valor do acumulador não for igual ao valor testado, o indicador de zero permanece desativado.

Se o valor do acumulador for menor que o valor que está sendo verificado, o indicador de transporte (C) do registro P permanecerá no estado desativado. Se o valor do acumulador for maior que o valor que está sendo verificado, o indicador de transporte será ativado.

CPX e CPY funcionam exatamente como CMP, exceto pelo fato de compararem valores com o conteúdo dos registros X e Y. Os efeitos serão os mesmos nos indicadores de status do registro P.

USANDO EM CONJUNTO AS INSTRUÇÕES DE COMPARAÇÃO E DE DESVIO

As três instruções de comparação da linguagem assembly Apple são geralmente usadas em conjunto com outras oito instruções – as instruções de desvio condicional mencionadas no Capítulo 6.

O programa-exemplo BULLETIN contém uma instrução de desvio condicional na linha 20. Essa instrução é a BNE LOOP, que significa "desviar para o *loop* cujo nome aparece no comando, se o indicador de zero (do registro de status do processador) não estiver ativado". Lembre-se de que o indicador de zero é *ativado* (igual a 1) se o resultado de uma operação recém-executada for 0, e é *desativado* (igual a 0) se o resultado da operação for *diferente de 0*.

Quando o computador encontrar a instrução BNE LOOP na linha 20, desviará sempre para a linha 16 (intitulada LOOP) até que o valor do registro X tenha sido decrementado para 0.

Quando isto acontecer, o comando BNE LOOP da linha 20 será ignorado, e o programa irá para a linha seguinte. Essa linha contém uma instrução RTS, que termina o programa.

INSTRUÇÕES DE DESVIO CONDICIONAL

São oito as instruções de desvio condicional na linguagem assembly 6502B/65C02. Todas começam com a letra B e são, também, chamadas instruções de endereçamento relativo ou instruções de desvio relativo. A Tabela 8-1 apresenta estas oito instruções e seus respectivos significados.

AS DIFERENÇAS ENTRE DESVIO E SALTO ("JUMP")

Existe outra categoria de instruções do 6502/6502B/65C02, representada pelas instruções de salto (jumps), que apresentam algumas diferenças importantes com relação às instruções de desvio.

São duas as instruções de salto na linguagem assembly 6502: JMP e JSR. O mnemônico JMP é usado de forma muito similar à da instrução GOTO em BASIC: quando encontra uma instrução JMP, o programa salta para o endereço de memória especificado pelo operando da instrução JMP.

A instrução JSR é semelhante à GOSUB do BASIC. Quando encontrada em um programa, o endereço da próxima instrução do programa é armazenado na pilha de hardware. O programa, então, salta para o endereço de memória especificado pelo operando da instrução JSR.

O mnemônico JSR é projetado basicamente para o uso com sub-rotinas. Na linguagem assembly 6502/6502B/65C02, as sub-rotinas sempre terminam com uma instrução RTS.

Tabela 8-1 Instruções de Desvio Condicional

BCC –	Branch if the carry (C) flag of the processor status (P) register is clear.	Desviar se o indicador de transporte (C) do registro de status do processador (P) estiver desativado. (Se estiver ativado, a operação não terá efeito.)
BCS –	Branch if the carry (C) flag is set.	Desviar se o indicador de transporte (C) estiver ativado. (Se estiver desativado, a operação não tem efeito.)
BEQ –	Branch if the result of an operation is zero (if the zero [Z] flag is set)	Desviar se o resultado de uma operação é zero (ou seja, se o indicador de zero [Z] estiver ativado).
BMI –	Branch on minus (if an operation results in a set N [negative] flag).	Desviar se negativo (se a operação resultar em ativamento do indicador de negativo [N]).
BNE –	Branch if not equal to zero (if the zero [Z] flag is not set).	Desviar se diferente de zero (i.é., se o indicador de zero [Z] não estiver ativado).
BPL –	Branch on plus (if an operation results in a cleared negative [N] flag).	Desviar se positivo (se a operação resulta em desativamento do indicador de negativo [N]).
BVC –	Branch if the overflow (V) flag is clear.	Desviar se o indicador de overflow (V) estiver desativado.
BVS –	Branch if the overflow (V) flag is set.	Desviar se o indicador de overflow (V) estiver ativado.

Em linguagem assembly, RTS é exatamente o oposto de JSR. Quando uma instrução RTS é encontrada em um programa, um endereço de memória é removido da pilha, e o processamento imediatamente salta para esse endereço. Se a instrução RTS foi usada para terminar uma sub-rotina, o endereço retirado da pilha é, normalmente, o que nela foi depositado pela instrução JSR usada para chamar a sub-rotina. Desta forma, o processamento do programa continuará na linha subseqüente à da instrução JSR.

Como vimos, as instruções de desvio são condicionais; as instruções de salto, entretanto, são *incondicionais*. Quando encontrada em um programa, uma instrução de salto sempre será realizada, enquanto no caso da instrução de desvio, ela somente é realizada quando determinadas condições são satisfeitas.

Existe outra importante diferença entre as instruções de desvio e as instruções de salto. Em linguagem de máquina, o operando de uma instrução de salto é sempre expresso por um valor de 2 bytes e é sempre interpretado como o endereço real de destino da instrução de salto. No caso da instrução de desvio, quando traduzida em linguagem de máquina, o operando é sempre convertido em um número de 1 byte, com sinal. Quando o programa é executado, esse número é interpretado como um *deslocamento* (offset) que aponta o endereço de início da destinação do desvio.

Vejamos um exemplo de comando que contém uma instrução de salto:

```
JMP $C000
```

Caso este comando fosse convertido em linguagem de máquina e executado, o resultado seria direto: o valor $C000 seria carregado no contador de programa e ocorreria um salto para o endereço de memória $C000.

Já as instruções de desvio são um pouco mais complicadas. O Programas 8-3 é um programa-exemplo que utiliza a instrução de desvio BCC, que significa "desviar se o transporte estiver ativado" (branch if carry set). O programa é intitulado BRANCHIT.S.

Programa 8-3
O PROGRAMA BRANCHIT.S
(Versão código-fonte)

```
 1   ORG $8000
 2 *
 3 WHAZIS EQU $0300
 4 *
 5   LDA #5
 6   CLC
 7   ADC WHAZIS
 8   BCS RETURN
 9   TAX
10 RETURN RTS
```

Este programa é uma rotina bastante simples: na linha 5, o número 5 é carregado no acumulador. A seguir, é desativado o indicador de transporte, e o valor armazenado no endereço de memória $0300 (ao qual foi dado o título WHAZIS) é adicionado ao valor armazenado no acumulador (no momento 5). Na linha 8, uma instrução de desvio é chamada.

Se a adição de 5 ao valor de WHAZIS resultar em um transporte (ou seja, se a soma de 5 e WHAZIS for maior que 255), a rotina desviará para a linha 10 e terminará. Entretanto, se a soma não resultar em um transporte, será transferida ao registro X antes do término da rotina.

COMPILANDO A ROTINA BRANCHIT.S

A Tabela 8-2 apresenta a listagem compilada do programa BRANCHIT.S. As linhas de 8 a 10 dessa tabela mostram como funciona a instrução de desvio. Na linha 8, à esquerda do número de linhas, vemos

8006:B0 01

O primeiro termo desta linha, $8006, é o endereço de memória no qual a instrução BCS será armazenada quando convertida em linguagem de máquina. O segundo termó, $B0, é o equivalente, em linguagem de máquina, da instrução BCS. O terceiro número, $01, é um valor de deslocamento que deve ser computado pelo 6502B ou 65C02, antes da realização da instrução BCS.

VALORES DE DESLOCAMENTO *(OFFSET)*

Em uma instrução de desvio, no 6502B/65C02, um valor de deslocamento é um número com sinal, que deve ser somado a um dado endereço de memória, para obtenção do endereço de destino da instrução de desvio.

O endereço ao qual o valor de deslocamento deve ser somado é sempre o endereço que *segue* o comando que contém a instrução de desvio. Assim, o deslocamento na iinha 8 do programa BRANCHIT.S é 1. Quando esse 1 é adicionado ao endereço da instrução seguinte – $8008 – a soma é $8009. Este número é o endereço da instrução RTS que termina o programa BRANCHIT.S.

Tabela 8-2 O Programa BRANCHIT.S (Versão Convertida)

8000:	1	ORG	$8000
8000:	2 *		
8000:	3 WHAZIS	EQU	$0300
8000:	4 *		
8000:A9 05	5	LDA	#5
8002:18	6	CLC	
8003:6D A7 02	7	ADC	WHAZIS
8006:B0 01	8	BCS	RETURN
8008:AA	9	TAX	
8009:60	10 RETURN	RTS	

Ao escrever uma instrução de desvio em linguagem assembly, pode-se tanto co-

locar um endereço, simplesmente, quanto um título que corresponda a um endereço. Quando o programa for compilado, o assembler converterá aquele endereço ou título em um valor de deslocamento. Sempre que encontrar uma instrução de desvio em um programa compilado, o 6502B/65C02, automaticamente, usará o deslocamento que segue a instrução para computar o endereço de destino do desvio.

Lembre-se de que o deslocamento que segue uma instrução de desvio nunca pode apresentar mais de 1 byte. Como este byte é sempre interpretado como um número com sinal, um deslocamento para desvio não pode ser menor que –128 ou maior que + 127.

Assim, uma vez que esse deslocamento é sempre adicionado ao endereço da primeira instrução que sucede a instrução de desvio, o deslocamento efetivo estará compreendido entre –126 e + 129.

Desta forma, os desvios que ocorrem como resultado de instruções de desvio estão sujeitos a certas limitações. o endereço de destino de uma instrução de desvio não pode estar a mais de 126 bytes abaixo ou a mais de 129 bytes acima do endereço da primeira instrução subseqüente à instrução de desvio.

E o que devemos fazer, então, se desejarmos um desvio para um endereço que não se enquadra nestes limites? Simplesmente usamos a instrução para desviar para uma instrução de salto, que não é sujeita a tais restrições. O Programa 8-4 exemplifica este procedimento.

Programa 8-4
O PROGRAMA BRANCHIT.S (COM A ADIÇÃO DE UMA INSTRUÇÃO DE SALTO)
(Com a Adição de uma Instrução de Salto)

```
 1   ORG $8000
 2 *
 3 WHAZIS EQU $0300
 4 *
 5   LDA #5
 6   CLC
 7   ADC WHAZIS
 8   BCC CONT
 9   JMP FARJMP ;(CAN BE ANYWHERE IN MEMORY)
10 CONT TAX
11   RTS
```

DESVIO A LONGA DISTÂNCIA

Na versão do programa BRANCHIT.S apresentada no Programa 8-4, a instrução BCS, existente no programa original, foi substituída por uma instrução BCC. Uma nova linha,

contendo uma instrução JMP – que pode saltar para qualquer endereço – foi inserida após a linha que contém a instrução BCC.

Nesta versão do programa, se a soma de 5 e o valor de WHAZIS resultar em um transporte, o programa saltará ao endereço intitulado FAR JMP, que pode corresponder a qualquer localização na memória. Não ocorrendo transporte, o programa pula para a linha 10, intitulada CONT ("continue") e procede como anteriormente.

COMO SÃO USADAS AS INSTRUÇÕES DE DESVIO CONDICIONAL

Vimos que a forma usual de utilização de uma instrução de desvio condicional em linguagem assembly 6502B/65C02 é a de carregar o registro X ou o Y com um 0 ou algum outro valor, e então carregar o registro A (ou um registro de memória) com um valor a ser usado para comparação.

A seguir, usamos uma instrução de desvio condicional para dizer ao computador quais indicadores do registro P devem ser verificados, e o que fazer se tais testes tiverem êxito ou não.

Compreendido o conceito básico do desvio condicional, pode-se utilizar uma tabela, como a Tabela 8-3, para escrever instruções de desvio.

Tabela 8-3 Uso das Instruções de Desvio Condicional

PARA VERIFICAR SE:	USE ISTO:	E DEPOIS ISTO:
A = VALOR	CMP# VALOR	BEQ
A <>VALOR	CMP# VALOR	BNE
A >= VALOR	CMP# VALOR	BCS
A >VALOR	CMP# VALOR	BEQ e a seguir BCS
A <VALOR	CMP# VALOR	BCC
A = (ENDEREÇO)	CMP $ENDEREÇO	BEQ
A <>ENDEREÇO	CMP $ENDEREÇO	BNE
A >= (ENDEREÇO)	CMP $ENDEREÇO	BCS
A >(ENDEREÇO)	CMP $ENDEREÇO	BEQ e a seguir BCS
A <(ENDEREÇO)	CMP $ENDEREÇO	BCC
X = VALOR	CPX# VALOR	BEQ
X <>VALOR	CPX# VALOR	BNE
X >= VALOR	CPX# VALOR	BCS
X >VALOR	CPX# VALOR	BEQ e a seguir BCS
X <VALOR	CPX# VALOR	BCC
X = (ENDEREÇO)	CPX $ENDEREÇO	BEQ
X <>(ENDEREÇO)	CPX $ENDEREÇO	BNE
X >= (ENDEREÇO)	CPX $ENDEREÇO	BCS
X >(ENDEREÇO)	CPX $ENDEREÇO	BEQ e a seguir BCS
X <(ENDEREÇO)	CPX $ENDEREÇO	BCC
Y = VALOR	CPY# VALOR	BEQ
Y <>VALOR	CPY# VALOR	BNE
Y >= VALOR	CPY# VALOR	BCS
Y >VALOR	CPY# VALOR	BEQ e a seguir BCS

"LOOPS" EM LINGUAGEM ASSEMBLY

Na linguagem assembly 6502B/65C02, as instruções de comparação e de desvio são, geralmente, utilizadas em conjunto. No programa BULLETIN, a instrução de comparação CPX e a de desvio BNE são usadas num loop controlado pela incrementação de um valor no registro X.

Cada vez que o *loop* do programa completa um ciclo, o valor do registro X é progressivamente incrementado. Cada vez que o programa chega à linha que contém a instrução INX, o valor do registro X é comparado com o número 29. Atingido esse número, o *loop* termina.

O programa, desta forma, continuará sempre voltando à linha que contém o comando JSR COUT, até que 29 caracteres tenham sido apresentados na tela.

O PROGRAMA BULLETIN.B

Estamos prontos, agora, para observar um novo programa, chamado BULLETIN.B. O Programa 8-5 provê a segunda metade da mensagem enigmática apresentada no programa BULLETIN original. Ele também contém alguns aperfeiçoamentos que o tornam mais versátil que seu predecessor – e mais fácil de se entender.

Programa 8-5
BULLETIN. B
(Versão Merlin Pro Assembler)

```
1 *
```

```
 2 * BULLETIN.B
 3 *
 4  ORG $8000
 5 *
 6 EOL EQU 13
 7 BUFLEN EQU 40
 8 FILLCH EQU $20
 9 COUT EQU $FDED
10 *
11  JMP START
12 *
13 TEXT ASC "A DOGMA GOT HIT BY A KARMA"
14  DB EOL
15 *
16 * LIMPAR BUFFER DE TEXTO
17 *
18 START LDA #FILLCH
19  LDX #BUFLEN
20 STUFF DEX
21  STA TXTBUF,X
22  BNE STUFF
23 *
24 * ARMAZENAR MENSAGEM NO BUFFER
25 *
26  LDX #0
27 LOOP1 LDA TEXT,X
28  STA TXTBUF,X
29  CMP #EOL
30  BEQ PRINT
31  INX
32  CPX #BUFLEN
33  BCC LOOP1
34 *
35 * APRESENTAR MENSAGEM
36 *
37 PRINT LDX #0
38 LOOP2 LDA TXTBUF,X
39  PHA
40  JSR COUT
41  PLA
42  CMP #EOL
43  BNE NEXT
44  JMP FINI
45 NEXT INX
46  CPX #BUFLEN
47  BCC LOOP2
48 *
49 FINI RTS
50 *
51 TXTBUF DS BUFLEN
```


O Programa 8-5 foi criado com um Merlin Pro, mas, com pequenas modificações, funcionará também com um sistema Apple ProDOS Assembler. O Programa 8-6 é a versão ORCA/M.

Programa 8-6
BULLETIN. B
(Versão ORCA/M Assembler)

```
          KEEP BULLETIN

MAIN      START

EOL       EQU   13
BUFLEN    EQU   40
FILLCH    EQU   $20
COUT      EQU   $FDED
          LDA   #FILLCH
          LDX   #BUFLEN
STUFF     DEX
          STA   TXTBUF,X
          BNE   STUFF
          LDX   #0
LOOP1     LDA   TEXT,X
          STA   TXTBUF,X
          CMP   #EOL
          BEQ   PRINT
          INX
          CPX   #BUFLEN
          BCC   LOOP1
PRINT     LDX   #0
LOOP2     LDA   TXTBUF,X
          PHA
          JSR   COUT
          PLA
          CMP   #EOL
          BNE   NEXT
          JMP   FINI
NEXT      INX
          CPX   #BUFLEN
          BCC   LOOP2

FINI      RTS

TEXT      DC    C'A DOGMA GOT HIT BY A KARMA'
          DC    H'0D'

TXTBUF    DS    40

          END
```

Como se pode ver, o programa BULLETIN.B é bastante similar ao original. Após digitar, compilar e rodar o programa, você verá que ele realiza essencialmente as mesmas operações que o BULLETIN, mas de forma um pouco mais elegante. A diferença mais óbvia entre os dois programas é a forma de manuseio das seqüências de texto.

O BULLETIN original utilizava uma seqüência de texto composta de códigos ASCII. No BULLETIN.B, não há a necessidade de converter seqüências de texto em números-código ASCII. Os três assemblers usados para criar os programas deste livro executam essa função, automaticamente.

Outra difereça importante entre o BULLETIN.B e seu predecessor é a forma pela qual é escrito o *loop* que lê os caracteres. No BULLETIN, o *loop* contava os caracteres até que essa contagem atingisse 29, quando, então, terminava. Este é um bom sistema para apresentação de seqüências de texto de 29 caracteres. Porém, não servirá para apresentar seqüências de outros comprimentos, não sendo portanto uma rotina versátil de apresentação de caracteres.

PROCURANDO UM<Return>

O BULLETIN B. é uma rotina mais versátil, já que pode apresentar seqüências quase que de qualquer extensão. O programa não controla o número de caracteres já apresentados através de contagem. Pelo contrário, ele, ao encontrar um caractere, verifica se o seu valor é $0D – o código ASCII para o retorno de carro (carriage return) ou caractere de fim de linha (EOL = end of line).

Se o caractere não é um EOL, o computador o apresenta e verifica o próximo caractere da seqüência. Se este for um EOL, o computador o apresenta na tela, e a rotina termina.

Outra diferença entre o BULLETIN e o BULLETIN.B é que o último não lê caracteres e os apresenta em um único passo: os caracteres são, primeiramente, colocados em um armazenador (buffer) e depois o conteúdo do buffer é apresentado na tela.

Buffers de texto são, geralmente, usados em programas em assembly, devido à versatilidade e facilidade de utilização. O texto pode ser carregado em um buffer a partir do teclado, por exemplo, ou por um modem telefônico, ou até mesmo diretamente da memória do computador.

Estando no buffer, uma seqüência pode ser removida de várias maneiras diferentes – não importando como os caracteres tenham sido ali colocados ou quais são eles.

Assim, uma vez que algumas sub-rotinas tenham sido escritas e preenchido um buffer, podem ser usadas para diferentes finalidades. Um buffer pode servir, desta forma, como um depósito central de seqüências de texto, tornando-as facilmente acessíveis.

"LIMPANDO" UM BUFFER DE TEXTO

Antes de se utilizar um buffer de texto é conveniente retirar quaisquer caracteres remanescentes, de forma que foi incluída, no programa BULLETIN.B, uma rotina de "limpeza" de buffer.

É uma rotina simples, que limpará um buffer de texto – ou qualquer outro bloco de memória que não exceda as limitações de comprimento intrínsecas à rotina – e irá preenchê-lo com espaços, zeros ou qualquer outro valor escolhido. No programa BULLETIN.B, a rotina preencherá o buffer com espaços, que assim aparecerão na tela.

Conforme for trabalhando com linguagem assembly, você descobrirá que rotinas de limpeza de memória como esta são de utilidade em diversos tipos de programa. Processadores de texto, programas de telecomunicação e vários outros tipos de software utilizam amplamente rotinas que retiram valores de blocos de memória e os substituem por outros valores.

A rotina de limpeza de memória, existente no programa BULLETIN.B, usa endereçamento indexado (direto) e um contador regressivo no registro X. Ela preencherá cada endereço de memória em um buffer de texto (TXTBUFF) com o "caractere de preenchimento" (FILLCH) designado, e então terminará.

Essa rotina trabalhará com qualquer caractere de 8 bits e com comprimentos de buffer (BUFLEN) de até 255 caracteres. Mais adiante, serão também apresentadas rotinas de 16 bits, que podem preencher blocos maiores de RAM com valores designados.

MAIS UM PROGRAMA: O JOGO DO NOME

O último programa deste capítulo utiliza muitas das técnicas de programação que até agora estudamos. O Programa 8-7 foi escrito em um Apple IIe, utilizando-se um Merlin Pro Assembler. Com pequenas modificações, poderá também ser escrito, compilado e rodado num Apple ProDOS Assembler. E, caso você possua um sistema ORCA/M, já deverá estar apto, a esta altura, a fazer as modificações necessárias para a utilização.

Program 8-7
THE NAME GAME

```
 1 *
 2 *  O JOGO DO NOME
 3 *
 4   ORG $8000
 5 *
 6 EOL  EQU $0D  ;TECLA RETURN
 7 EOF  EQU $03
 8 FILLCHR EQU $20  ;TECLA ESPACO
 9 BUFLEN EQU 40
10 GETLN1 EQU $FD6F  ;ROTINA PARA RECEBER UMA LINHA DE TEXTO DO
   TECLADO
11 COUT EQU $FDED  ;ROTINA PARA APRESENTAR UM CARACTERE NA TELA
   SCREEN
12 TEMPTR EQU $FB
13 OSBUF EQU $200
14 *
15   JMP BEGIN
16 *
17 COUNT DS 1
18 INPBUF DS 80
19 *
20 TITLE ASC  "O JOGO DO NOME"
21   HEX 0D
22 HELLO ASC "OLA,"
23   HEX 03
24 QUERY ASC "QUAL E O SEU NOME?"
25   HEX 0D
26 NAME ASC  "GEORGE"
27   HEX 0D
28 REBUFF ASC  "VA EMBORA,"
29   HEX 03
30 DEMAND ASC  "TRAGA-ME GEORGE!"
31   HEX 0D
32 GREET ASC  "OI, GEORGE!"
33   HEX 0D
34 *
35 * LIMPAR BUFFER DE TEXTO
36 *
37 FILL LDA #FILLCHR
38   LDX #BUFLEN
39 FILLOOP DEX
40   STA INPBUF,X
41   BNE FILLOOP
42   RTS
43 *
44 PRINT LDY #0
45 SHOW LDA (TEMPTR),Y
```



```
46   CMP #EOF
47   BEQ DONE
48   PHA
49   JSR COUT
50   PLA
51   CMP #EOL
52   BNE NEXT
53   JMP DONE
54 NEXT INY
55   CPY #BUFLEN
56   BCC SHOW
57 DONE RTS
58 *
59 BEGIN LDA #0
60   STA $COOA ;LIGAR FIRMWARE DE 80 COLUNAS
61   JSR $C300 ;TRANSFERIR O CONTROLE AO CARTAO DE 80 COLUNAS
62   STA $COOD ;ATIVAR DISPLAY EM 80 COLUNAS
63   JSR $FCS8 ;LIMPAR TELA E COLOCAR CURSOR NA POSICAO 'HOME'
64 *
65 *  PRINT "O JOGO DO NOME"
66 *
67   LDA #EOL
68   JSR COUT
69   LDA #<TITLE
70   STA TEMPTR
71   LDA #>TITLE
72   STA TEMPTR+1
73   JSR PRINT
74   LDA #EOL
75   JSR COUT
76 *
77 *  PRINT "OLA..."
78 *
79   LDA #<HELLO
80   STA TEMPTR
81   LDA #>HELLO
82   STA TEMPTR+1
83   JSR PRINT
84 *
85 *  PRINT 'QUAL E O SEU NOME?'
86 *
87 ASK LDA #<QUERY
88   STA TEMPTR
89   LDA #>QUERY
90   STA TEMPTR+1
91   JSR PRINT
92   LDA #EOL
93   JSR COUT
94 *
95 *  INTRODUZIR UMA LINHA DIGITADA
```

```
 96 *
 97  JSR GETLN1
 98  STX COUNT
 99  LDY #0
100 PRLOOP LDA OSBUF,Y
101  STA INPBUF,Y
102  INY
103  DEX
104  BNE PRLOOP
105 *
106 *  VERIFICAR SE O NOME E GEORGE
107 *
108  LDX COUNT
109  CPX #6
110  BNE NOGOOD
111  DEX
112 CHECK LDA INPBUF,X
113  CMP NAME,X
114  BNE NOGOOD
115  DEX
116  BNE CHECK
117  JMP DUNIT
118 *
119 *  NAO; PRINT 'VA EMBORA...'
120 *
121 NOGOOD LDA #EOL
122  JSR COUT
123  LDA #<REBUFF
124  STA TEMPTR
125  LDA #>REBUFF
126  STA TEMPTR+1
127  JSR PRINT
128 *
129 *  PRINT NOME DO JOGADOR
130 *
131  LDA #<INPBUF
132  STA TEMPTR
133  LDA #>INPBUF
134  STA TEMPTR+1
135  JSR PRINT
136  LDA #EOL
137  JSR COUT
138 *
139 *  PRINT 'E TRAGA-ME GEORGE!'
140 *
141  LDA #<DEMAND
142  STA TEMPTR
143  LDA #>DEMAND
144  STA TEMPTR+1
145  JSR PRINT
```



```
146  LDA #EOL
147  JSR COUT
148  JMP ASK
149 *
150 *  SIM; PRINT CUMPRIMENTO
151 *
152 DUNIT JSR $FBDD ;SOUND A BEEP
153  LDA #EOL
154  JSR COUT
155  LDA #<GREET
156  STA TEMPTR
157  LDA #>GREET
158  STA TEMPTR+1
159  JSR PRINT
160  RTS
```

Se você digitou, compilou e executou os programas BULLETIN e BULLETIN.B, não deverá ter dificuldade em entender o funcionamento do programa O JOGO DO NOME. Usando várias sub-rotinas bem simples, ele apresenta uma mensagem na tela e aguarda uma resposta. Se o programa considerar sua resposta incorreta, pedirá que tente novamente. Quando, finalmente, você digitar a linha pela qual o programa está procurando, receberá uma mensagem de "recompensa" e o programa terminará.

A ROTINA GETLN1

Além da importante rotina COUT, usada nos programas BULLETIN e BULLETIN.B, destinada à apresentação de caracteres na tela, o programa O JOGO DO NOME utiliza uma rotina residente chamada GETNL1, que *lê* linhas de texto introduzidas a partir do teclado.

O endereço de chamada da rotina GETLN1 é $FD6F, como se pode constatar observando a linha 10 do programa. Para usar a rotina GETLN1 em um programa em assembly, listamos o seu endereço na tabela de símbolos do programa, e emitimos um comando como

```
JSR GETLN1
```

Este comando pode ser observado na linha 97 do programa. Ao rodar o programa, é nesta linha que o computador parará e aguardará a digitação de um nome. Assim que este for digitado e que for pressionada a tecla <return>, o computador armazenará a linha digitada em um buffer especial, que começa no endereço de memória 200. O número de caracteres desta linha recém-digitada é colocado no registro X do 6502B/65C02.

Na tabela de símbolos de O JOGO DO NOME, o buffer que inicia no registro de memória 200 é chamado OSBUF. O conteúdo deste buffer pode mudar rapidamente; assim, tão logo este buffer é usado no programa, seu conteúdo é copiado em outro buffer, chamado INPBUF (este procedimento é realizado nas linhas de 98 a 104).

Nas linhas de 108 a 117, é efetuada uma verificação, para saber se o nome digitado em OSBUF é George. Se não for, o programa salta para uma rotina que pede para ver o George. Se o nome for George, o computador responde com um *bip* e um cumprimento.

USANDO O DISPLAY DE 80 COLUNAS DO APPLE

As linhas 59 e 60 do programa inicializam o firmware que permite ao Apple IIc/IIe gerar uma tela em 80 colunas. Na linha 61, o controle é transferido ao firmware de 80 colunas, com o comando JSR$C300, que executa a mesma tarefa que o comando PR#3 do BASIC.

Na linha 62, o display em 80 colunas é realmente ativado, com o comando STA $C00D. Finalmente, na linha 63, o programa chama uma rotina residente que inicia em $FC58. Esta rotina limpa a tela, colocando o cursor no seu canto superior esquerdo.

O detalhamento completo do funcionamento destas rotinas pode ser encontrado nos manuais de consulta (reference manuals) do Apple IIc e do Apple IIe. Se, entretanto, você apenas desejar saber como usar a característica de 80 colunas de seu computador em programas em assembly, as linhas de 59 a 63 do programa O JOGO DO NOME contém tudo o que é necessário.

SÍMBOLOS "BYTE INFERIOR" E "BYTE SUPERIOR"

A técnica usada em O JOGO DO NOME para armazenar números de 16 bits em localizações de memória de 8 bits, como byte inferior e byte superior, merece especial atenção. Esta técnica aparece pela primeira vez nas linhas de 69 a 72.

```
69      LDA     #<TITLE
70      STA     TEMPTR
71      LDA     #>TITLE
72      STA     TEMPTR+1
```

Você deve ter percebido o que esta seqüência faz. No código-fonte reconhecido pela maioria dos assemblers, a seqüência "#<HELLO" significa "o byte inferior do endereço intitulado 'HELLO' e a seqüência "#>HELLO" significa "o byte superior do endereço intitulado 'HELLO' ". (Esta seqüência não se refere ao conteúdo do endereço, mas ao próprio endereço, uma vez que, na linguagem assembly 6502/6502B/65C02, o símbolo "#" é usado para identificar um número propriamente dito.)

As linhas de código acima primeiramente armazenam o byte de ordem inferior do endereço de 16 bits da seqüência "HELLO" na localização de memória intitulada "TEMPTR" (a qual, como se pode constatar na tabela de símbolos do programa, é o endereço de memória $FB). A seguir, armazenam o byte de ordem superior da seqüência "HELLO" na localização de memória TEMPTR+1 ou $FC.

JOGANDO O JOGO DO NOME

Após chamar à ação o firmware de 80 colunas, o programa apresenta seu título na tela. As duas próximas rotinas apresentam a linha "OLÁ! QUAL É O SEU NOME?"

A seguir, o programa limpa o buffer de texto e espera até que você digite uma resposta. Ao fazer isto, você estará colocando cada um dos caracteres no buffer de texto. Este processo continuará até que você tecle um <return>.

O programa, então, examina os caracteres introduzidos, verificando se eles formam o nome George. Se isto não acontece, o programa pedirá que lhe tragam George. Quando, finalmente, for digitado o nome George, o computador emitirá um bit e proclamará: "OLA!, GEORGE!".

Você verá que os princípios usados para criar as rotinas de entrada e saída deste programa são utilizados em muitos programas em linguagem assembly.

CAPÍTULO 9

MANIPULAÇÃO BIT A BIT DOS NÚMEROS BINÁRIOS

Existem 65.536 bytes de memória em um computador Apple IIe sem expansão, e 131.072 bytes num Apple IIc ou num IIe expandido. Como há oito bits em cada byte, temos 524.288 bits em um Apple IIe normal e 1.048.576 bits num Apple IIe de 128K ou num Apple IIc normal.

O que isto significa para o programador em linguagem assembly? Se você sabe como realizar operações com um único bit em números binários, pode então controlar cada bit em seu Apple – um grande controle sobre um computador.

No primeiro capítulo deste livro, vimos como controlar um dos bits mais importantes no microprocessador: o bit de transporte (carry bit) do 6502B/65C02. A manipulação do bit de transporte é uma das mais importantes técnicas de manipulação de bits em linguagem assembly. Obtivemos, também, bastante experiência na utilização do bit de transporte em programas de adição.

USANDO O BIT DE TRANSPORTE EM OPERAÇÕES DE DESLOCAMENTO DE BITS

Você sabe que o microprocessador 6502B/65C02 de seu Apple é um *chip* de 8 bits: ele não realiza operações com números maiores que 255 sem que o faça por etapas.

O 6502B/65C02 tem de quebrar um número maior em "pedaços" de 8 bits e realizar as operações com cada parte do número. A seguir, o número tem de ser "reconstruído".

Familiarizados com este processo, veremos que ele não é tão complicado como parece. De fato, o "material" usado para este corta-e-cola eletrônico está contido em um único bit – o bit de transporte do registro P.

INSTRUÇÕES DE DESLOCAMENTO DE BITS

Vimos como funcionam as operações de transporte em diversos programas, neste livro. Para termos uma visão mais clara do funcionamento do transporte na aritmética 6502B/65C02, entretanto, será interessante examinarmos quatro instruções, em linguagem de máquina, muito especializadas: ASL (arithmetic shift left = deslocamento aritmético à esquerda), LSR (logical shift right = deslocamento lógico à direita), ROL (rotate left = rotação à esquerda) e ROR (rotate right = rotação à direita). Estas quatro instruções são extensivamente utilizadas em linguagem assembly 6502B/65C02.

ASL (ARITHMETIC SHIFT LEFT = DESLOCAMENTO ARITMÉTICO À ESQUERDA)

Como você deve lembrar do Capítulo 2 (sobre aritmética binária), cada número que termina em 0, em notação binária, é o dobro do número binário precedente que termina em 0. Por exemplo, 1000 0000 ($80) é o dobro de 0100 0000 ($40), que é o dobro de 0010 0000 ($20), que por sua vez é o dobro de 0001 0000 ($10).

Desta forma, é muito fácil multiplicar um número binário por 2. Basta deslocar cada bit do número, uma casa à esquerda, e colocar um 0 no lugar que ficou vazio devido ao deslocamento (o bit 0, ou o bit mais à direita, do número). Se o bit 7 (o bit mais à esquerda) do número a ser dobrado é 1, então deve-se providenciar um transporte.

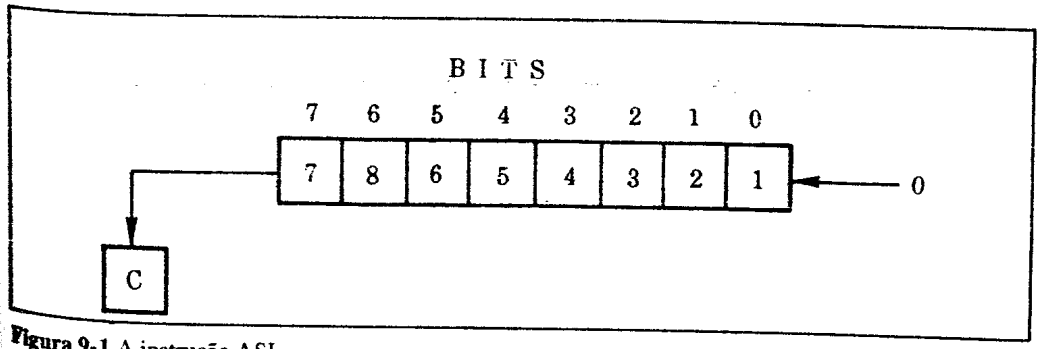


Figura 9-1 A instrução ASL.

A operação toda que descrevemos – deslocar um byte à esquerda, com transporte – pode ser realizada com uma única instrução em linguagem assembly: ASL. Como ilustra a Figura 9-1, a instrução ASL move cada bit, em um número de 8 bits, uma casa para a esquerda. E o bit 7 vai para o bit de transporte do registro de status do processador (P).

A instrução ASL apresenta vários usos na linguagem assembly 6502B/65C02. Por exemplo, é uma forma fácil de multiplicar números por 2. O Programa 9-1 é uma rotina para dobrar números, da forma que se apresenta em um programa criado com um Merlin Pro ou Apple ProDOS.

Programa 9-1
DOBRANDO UM NÚMERO COM A INSTRUÇÃO ASL

```
1 *
2   ORG $8000
3 *
4   LDA #$40 ;REM 0100 0000
5   ASL A ; DESLOCAR VALOR NO ACUMULADOR PARA A ESQUERDA
6   STA $FB
7   RTS
```

Se você rodar o Programa 9-1 e, então, usar o monitor de linguagem de máquina para examinar o conteúdo do endereço de memória $FB, verá que o número $40 (0100 0000) foi dobrado para $80 (1000 0000) antes de ser armazenado no endereço $FB.

Outro uso da instrução ASL é a "compressão" (pack) de dados, que aumenta a capacidade efetiva da memória de um computador. Mais adiante, neste capítulo, será apresentado um exemplo de como comprimir dados usando a instrução ASL.

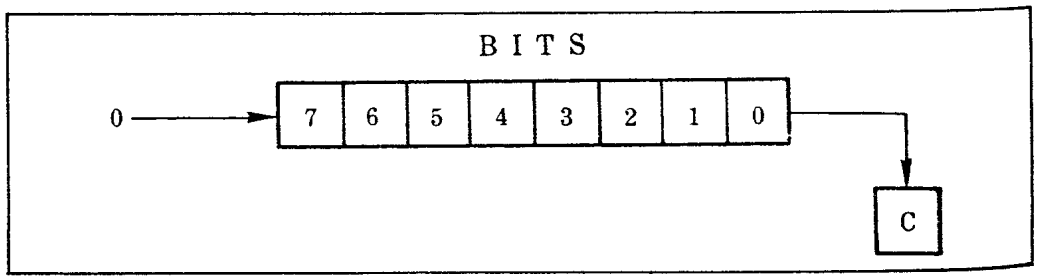


Figura 9-2 A instrução LSR.

LSR (LOGICAL SHIFT RIGHT = DESLOCAMENTO LÓGICO À DIREITA

Como mostra a Figura 9-2 , a instrução LSR é exatamente o oposto da instrução ASL.

Como Funciona a Instrução LSR A LSR, da mesma forma que a ASL, trabalha com qualquer número binário que esteja no acumulador do 6502B/65C02; porém, deslocará cada bit do número uma posição para a *direita*. O bit 7 do novo número ("esvaziado" pela instrução LSR) será preenchido com um 0, e o bit 0 será depositado no indicador de transporte do registro P.

Como ilustra o Programa 9-2, a instrução LSR pode ser usada para dividir um número par de 8 bits por 2.

Programa 9-2
DIVIDINDO UM NUMERO POR 2 COM A INSTRUCAO LSR

```
 1 *
 2 * LSRDIV
 3 *
 4 VALUE1 EQU $FB
 5 VALUE2 EQU $FC
 6 *
 7  ORG $8000
 8 *
 9  LDA#6; OU QUALQUER OUTRO NUMERO DE 8 BIT
10  STA VALUE1
11 *
12 *NOW WE'LL DIVIDE BY 2
13 *
14  LDA VALUE1
15  LSR A
16  STA VALUE2
17  RTS
```

Neste programa, o número armazenado em VALUE1 é dividido por 2, e o resultado armazenado em VALUE2. Esta rotina de divisão também informa se o número que foi dividido era par ou ímpar. Esta informação é deixada no bit de transporte do registro P: se a rotina deixa o bit de transporte desativado, o número dividido era par, se ativado, o número era ímpar.

Adiante, neste capítulo, veremos como "descomprimir" (unpack) dados.

Outro Teste para Par ou Ímpar O Programa 9-3 é outra rotina que pode determinar se um número é par ou ímpar. Nas linhas 12 e 13 da rotina ODDTEST, um registro de memória chamado FLAGADR (de *flag address*) é limpo, atribuindo-se a ele um 0. A seguir, o conteúdo do registro de memória VALUE1 é deslocado uma posição para a direita e armazenado em um terceiro registro, VALUE2. Se o valor que está sendo deslocado é par, a operação não ativa o bit de transporte, e a sub-rotina termina. Se o valor é ímpar, a operação ativa o bit de transporte, o programa pula para a linha 22, e o bit de transporte

ativado é rodado dentro do registro FLGADR, usando-se a instrução ROL (abordada logo adiante). Assim, se a rotina deixar um 0 em FLGADR, o número que foi dividido era par; se deixar um 1 em FLGADR, o número era ímpar.

Programa 9-3
A ROTINA ODDTEST

```
 1 *
 2 * ODDTEST
 3 *
 4 VALUE1 EQU $FB
 5 VALUE2 EQU $FC
 6 FLGADR EQU $FD
 7 *
 8  ORG $8000
 9 *
10  LDA #7 ;(ODD)
11  STA VALUE1
12  LDA #0
13  STA FLGADR ; LIMPANDO FLGADR
14 *
15  LDA VALUE1
16  LSR A ; REALIZAR A DIVISAO
17  STA VALUE2 ; EFETUADA
18 *
19  BCS FLAG
20  RTS ; TERMINA ROTINA SE TRANSPORTE DESATIVADO...
21 *
22 FLAG
24  ROL FLGADR
25  RTS ; ... E TERMINA O PROGRAMA
```

A utilização de LSR para "descomprimir" dados que foram "comprimidos" com ASL será discutida mais adiante, neste capítulo.

ROL E ROR

As instruções ROL (rotate left = rotação à esquerda) e ROR (rotate right = rotação à direita) são também usadas para o deslocamento de bits em número binários, mas elas utilizam o bit de transporte diferentemente das instruções AS e LSR. A Figura 9-3 ilustra a operação da instrução ROL.

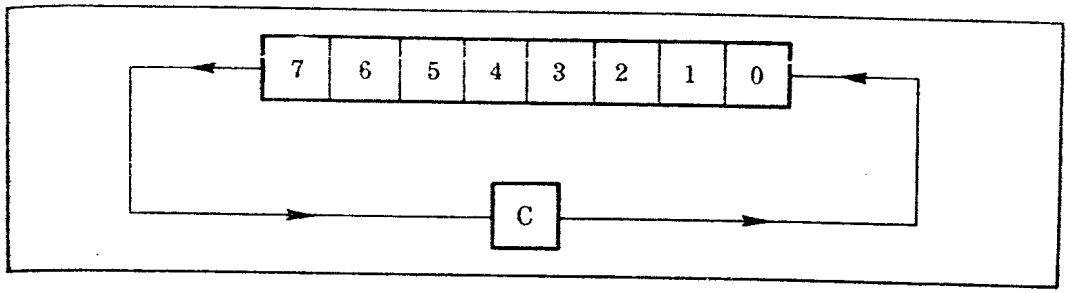


Figura 9-3 A instrução ROL.

ROL, da mesma forma que ASL, pode ser usada para deslocar o conteúdo do acumulador, ou de um registro de memória, uma casa para a esquerda. ROL, entretanto, não coloca um 0 na posição do bit 0 do número que está sendo deslocado para o bit de transporte. Em vez disso, coloca o bit de transporte no bit 0 do registro que está sendo deslocado, e então move cada um dos bits daquele registro uma casa para a esquerda, colocando então o bit 7 no bit de transporte. Se o bit de transporte estiver ativado quando isto acontece, um 1 é colocado na posição do bit 0 do byte que está sendo deslocado. Se o transporte estiver desativado, um 0 vai para a posição do bit 0 do registro deslocado.

A Figura 9.4 ilustra o uso da instrução ROR. Ela trabalha da mesma forma que a instrução ROL, mas em sentido oposto: movimenta cada bit do byte uma posição para a direita, e coloca o bit de transporte na posição do bit 7 do byte deslocado. O bit 0 do byte deslocado é movido para o bit de transporte do registro P.

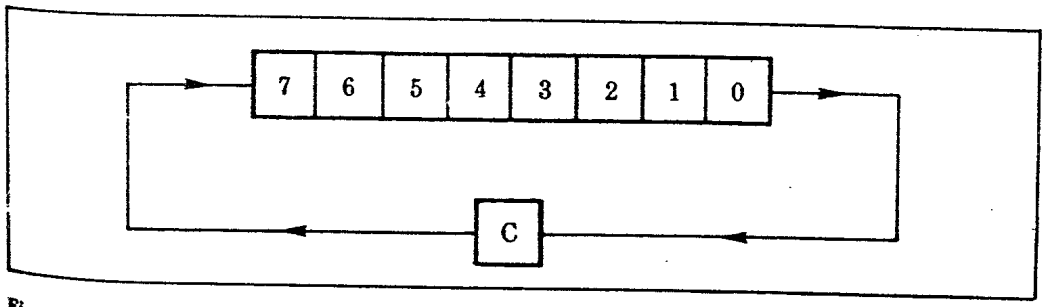


Figura 9-4 A instrução ROR.

ROL e ROR são freqüentemente usadas em rotinas 6502B/65C02 de multiplicação e divisão, bem como em outras rotinas nas quais os bits são deslocados e verificados.

OS OPERADORES LÓGICOS

Vejamos quatro importantes mnemônicos da linguagem assembly, chamados de *operadores lógicos*. São eles as instruções AND (e), ORA (ou), EOR (ou exclusivo) e BIT (bit).

Estes quatro operadores lógicos são usados para a comparação de valores. Trabalham, porém, de forma diferente da dos operadores de comparação CMP, CPX e CPY. Estes fornecem resultados muito gerais; podem determinar somente se dois valores são iguais e, se não, qual deles é o maior.

AND, ORA, EOR e BIT são instruções bem mais específicas, usadas para comparar bits individuais de números, apresentando assim larga utilização em programas em assembly para o Apple IIe/IIc.

LÓGICA BOOLEANA

Os quatro operadores lógicos da linguagem assembly utilizam-se dos princípios da ciência matemática chamados de *lógica booleana*. Em lógica booleana, os números binários 0 e 1 são usados não para expressar valores, mas para indicar se uma afirmação (ou comando) é falsa ou verdadeira. Se falsa, seu valor em lógica booleana diz-se ser 0; se verdadeira, diz-se ser 1.

Em linguagem assembly 6502B/65C02, o operador AND tem o mesmo significado da palavra *and* em Inglês (E).

Tabela 9.1 Tabela Verdade para AND (E)

0 AND 0 0	0 AND 1 0	1 AND 0 0	1 AND 1 1

Se um bit E outro bit têm valor 1 (e são, assim, "verdadeiros") o operador AND fornece um valor 1. Porém, se qualquer outra condição existe – se um bit é verdadeiro e o outro falso, ou se ambos são falsos – o operador AND retorna um 0 ou falso.

Os resultados de operadores lógicos são, geralmente, ilustrados sob a forma de diagramas chamados de *tabelas da verdade*. A Tabela 9.1 é a tabela da verdade para o operador AND.

Na linguagem assembly 6502B/65C02, a instrução AND é freqüentemente utilizada em uma operação chamada *mascaramento de bits* (bit mask-ing), cuja finalidade é a de ativar ou desativar bits específicos em um número. O operador AND pode ser usado, por exemplo, para desativar qualquer número de bits, colocando um 0 em cada bit a ser desativado:

```
100   LDA #AA ;BINARIO  1010 1010
110   AND #F0 ;BINARIO  1111 0000
```

Se o computador encontrar estas linhas em um programa, realizará a seguinte operação AND:

```
      1010 1010 (conteúdo do acumulador)
AND   1111 0000
      1010 0000 (novo valor no acumulador)
```

Neste exemplo, a instrução AND limpa o "nybble" inferior de $AA para $0 (com um resultado $A0). A mesma técnica servirá para qualquer outro número de 8 bits: não importa qual seja o número que passe através da máscara 1111 0000, seu nybble inferior será sempre trazido a $00, e seu nybble superior sempre sairá intacto da operação AND.

O OPERADOR ORA

Quando a instrução ORA (ou) é usada para comparar um par de bits, o resultado da comparação é 1 (verdadeiro) se o valor de *qualquer um* dos bits é 1. A Tabela 9.2 é a tabela da verdade para a instrução ORA.

ORA também é utilizada em operações de mascaramento de bits. Eis um exemplo de rotina de mascaramento com esta instrução:

Tabela 9.2 Tabela Verdade para ORA (ou)

0	0	1	1
ORA 0	ORA 1	ORA 0	ORA 1
0	1	1	1

```
LDA VALUE
ORA #$0F
STA DEST
```

Se o número em VALUE fosse $22 (binário 0010 0010), ocorreria a seguinte operação de mascaramento:

```
    0010 0010  (no acumulador)
ORA 0000 1111  (#$0F)
-------------
    0010 1111  (novo valor no acumulador)
```

O OPERADOR EOR

A instrução EOR (ou exclusivo) retornará um valor verdadeiro (1) se um – e apenas um – dos bits do par for 1. A Tabela 9.3 é a tabela da verdade para o operador EOR.

A instrução EOR é, geralmente, usada para comparar bytes, determinando se são idênticos. Se algum dos bits dos dois bytes que estão sendo comparados for diferente, o resultado da comparação é diferente de zero:

Exemplo 1

```
    1011 0110
EOR 1011 0110
    0000 0000
```

Exemplo 2

```
    1011 0110
EOR 1011 0111
    0000 0001
```

No Exemplo 1, os bytes comparados são idênticos; portanto, o resultado da comparação é 0. No Exemplo 2, um bit é diferente, e o resultado é diferente de zero.

O operador EOR é também usado para complementar valores. Se executarmos um EOR entre um valor de 8 bits e $FF, cada bit 1 do valor será complementado para 0 e cada bit 0 será complementado para 1:


```
    1110 0101  (no acumulador)
EOR 1111 1111
    0001 1010  (novo valor no acumulador)
```

Outra característica útil da instrução EOR é a execução, duas vezes, em um número, com o mesmo operando, o que transforma o número em um outro e depois restaura o valor original:

Tabela 9.3 Tabela Verdade para EOR

0	0	1	1
EOR 0	EOR 1	EOR 0	EOR 1
0	1	1	0

```
    1110 0101  (no acumulador)
EOR 0101 0011
    1011 0110  (novo valor no acumulador)
EOR 0101 0011  (mesmo operando acima)
    1110 0101  (valor original restaurado no acumulador)
```

Esta capacidade da instrução EOR é freqüentemente utilizada em gráficos de alta resolução, para colocar uma imagem sobre outra, sem destruir a que foi superposta.

COMPRIMINDO E DESCOMPRIMINDO DADOS NA MEMÓRIA

Vamos, agora, discutir a "compressão e descompressão" de dados, usando as instruções de deslocamento e de verificação de bits. Primeiramente, vejamos algo a respeito de como se pode comprimir dados para preservar espaço na memória do computador.

Para ter uma idéia do funcionamento da compressão de dados, suponha que você tem uma série de valores de 4 bits armazenada em um bloco de memória de seu computador.

Esses valores podem ser caracteres ASCII, números BCD (abordados mais adiante) ou qualquer outro tipo de valores de 4 bits.

Usando a instrução ASL, você pode comprimir tais valores, dois deles em cada byte do bloco de memória em que estão armazenados, ocupando, assim, metade do espaço que ocuparam antes da compressão.

O Programa 9.4 é uma rotina que pode ser utilizada em um *loop* para comprimir cada byte de dados

Programa 9.4
COMPRIMINDO DADOS COM A INSTRUÇÃO ASL

```
 1 *
 2 *PACKDATA
 3 *
 4  ORG $8000
 5 *
 6  NYB1 EQU $FB
 7  NYB2 EQU $FC
 8  PKDBYT EQU $FD
 9 *
10  LDA #$04  ; OU QUALQUER OUTRO VALOR DE 4 BITS
11  STA NYB1
12  LDA #$06  ; OU QUALQUER OUTRO VALOR DE 4 BITS
13  STA NYB2
14 *
15  CLC
16  LDA NYB1
17  ASL A
18  ASL A
19  ASL A
20  ASL A
21  ORA NYB2
22  STA PKDBYT
23  RTS
```

COMO FUNCIONA

A rotina do Programa 9.4 carrega um valor de 4 bits no acumulador, desloca esse valor para o nybble superior no acumulador, e (usando o operador lógico ORA) coloca outro valor de 4 bits no nybble inferior do acumulador. São comprimidos no acumulador, portanto, dois valores de 4 bits – e estes dois valores serão, então, armazenados em "PKDBYT", um único registro de memória de 8 bits.

VERIFICANDO O RESULTADO

Digite o programa no computador e execute-o, usando o monitor de linguagem de máquina de seu assembler. Após rodar o programa, você pode verificar (peek) a memória do computador para saber o que foi realizado, usando o monitor residente do Apple. Simplesmente chame o monitor de linguagem de máquina e digite o comando

```
FB.FD
```

seguido de um<return>. O computador responderá com a seguinte linha:

```
00FB- 04 06 46
```

Esta linha lhe diz que o número $04 foi armazenado no endereço de memória $FB e que o número $06 foi armazenado no endereço de memória $FC. Ambos foram comprimidos no endereço $FC.

Este exemplo mostra como a compressão de dados pode aumentar a capacidade efetiva de memória. Suponha ter um longo documento, composto somente de caracteres ASCII, que podem ser armazenados na memória na forma de números de 4 bits. Este texto, comprimido, ocupará somente a metade do espaço que ocuparia, já que é possível armazenar dois caracteres em cada registro de 8 bits da memória.

DESCOMPRIMINDO DADOS

Não existiria nenhuma vantagem na compressão de dados, é claro, se eles não pudessem ser descomprimidos mais tarde. Os dados comprimidos com o uso da ASL podem ser descomprimidos através da instrução complementar LSR, em conjunto com o operador lógico AND. O Programa 9.5 é uma rotina que mostra como os dados podem ser descomprimidos com a instrução LSR.

Programa 9.5
DESCOMPRIMINDO DADOS COM A INSTRUÇÃO LSR

```
10 *
20 * UNPACKIT
30 *
40 PKDBYT EQU $FB
50 LOWBYT EQU $FC
```

```
60 HIBYT EQU $FD
70 *
80  ORG $8000
90 *
100  LDA #255 ;OU QUALQUER OUTRO VALOR DE 8 BITS
110  STA PKDBYT
120  LDA #0 ;LIMPAR LOWBYT E HIBYT
130  STA LOWBYT
140  STA HIBYT
150 *
160  LDA PKDBYT
170  AND #$0F ;BINARIO  0000 1111
180  STA LOWBYT
190  LDA PKDBYT
200  LSR A
210  LSR A
220  LSR A
230  LSR A
240  STA HIBYT
250  RTS
```

Esta rotina funciona como o Programa 9.4, mas em sentido contrário. No Programa 9.5, o acumulador é carregado com um byte de 8 bits, no qual foram comprimidos dois valores de 4 bits. Os quatro bits superiores deste byte comprimido são, então, "zerados" usando-se o operador lógico AND, e o nybble inferior do byte é armazenado em um registro de memória chamado LOWBYT.

A seguir, o acumulador é carregado por uma segunda vez com o byte comprimido. Desta vez, o byte é deslocado para a direita, quatro vezes, com a instrução LSR. Esta manobra resulta em um valor de 4 bits que é, finalmente, armazenado em um registro de memória chamado HIBYT.

O valor comprimido em PKDBYT foi, desta forma, desmembrado ou descomprimido em dois valores de 4 bits – um armazenado em LOWBYT e outro em HIBYT. Cada um dos números de 4 bits (que poderá representar um caractere ASCII ou qualquer outro valor de 4 bits) pode, agora, ser processado como uma entidade separada.

O OPERADOR BIT

O operador BIT é uma instrução um pouco mais complicada que as instruções AND, ORA e EOR.

A instrução BIT é usada para determinar se um valor armazenado em um registro de memória coincide com o valor armazenado no acumulador. A instrução pode ser usada somente com endereçamento absoluto ou endereçamento da página zero. Eis dois exemplos de formatos corretos para a instrução BIT:

```
BIT $02A7
BIT $FB
```

Quando a instrução BIT é usada em qualquer um destes formatos, uma operação lógica AND é realizada no byte sob teste. O *oposto* do resultado desta operação é, então, armazenado no indicador de zero (zero flag) do registro de status do processador. Em outras palavras, se todos os bits *ativados* do acumulador coincidem com os bits ativados do valor em questão, *nas mesmas posições*, o indicador Z é desativado. Se não há coincidência de bits ativados, o indicador Z é ativado. O Programa 9.6 ilustra o uso da instrução BIT.

Programa 9.6
USANDO A INSTRUÇÃO BIT

```
1  LDA #01
2  BIT $02A7
3  BNE MATCH
4  JMP NOGOOD
5  MATCH RTS
```

É efetuada uma verificação BIT no valor armazenado no registro $02A7. Se há coincidência com o valor do acumulador, o indicador Z do registro P é desativado, e o programa desvia para a linha intitulada MATCH (coincide). Se não há coincidência, o indicador Z é ativado, e o programa pulará para a rotina de nome NO GOOD.

O mnemônico BIT também realiza algumas outras funções. Quando a instrução BIT é utilizada, os bits 6 e 7 do valor que está sendo verificado são sempre depositados diretamente nos bits 6 e 7 do registro P. Esta informação é bastante útil, pois os bits 6 e 7 são indicadores muito importantes do registro de status do processador. O bit 6 é o indicador de overflow (V) e o bit 7, o de negativo (N).

Assim, a instrução BIT pode, também, ser usada como um método rápido de verificação dos bits 6 e 7 de qualquer valor de 8 bits. Se o bit 6 do valor sob teste está ativado, o indicador V do registro P também será ativado, e uma instrução BVC ou BVS pode ser utilizada para determinar o que acontecerá a seguir, no programa. Se o bit 7 do valor sob teste estiver ativado, o indicador N do registo P também o estará, e uma instrução BPL ou BMI pode ser usada para direcionar o resultado da rotina.

É importante notar que, durante o transcurso destas ações, o valor do acumulador e o da localização de memória sob teste permanecem inalterados. Desta forma, se desejar realizar uma operação lógica AND, sem que ocorram alterações nos valores do acumulador e do registro a ser verificado, o mnemônico BIT deverá ser a melhor opção.

CAPÍTULO 10

A MATEMÁTICA DA LINGUAGEM ASSEMBLY

Neste capítulo você saberá como seu Apple soma, subtrai, multiplica e divide. O Apple IIc/IIe pode manusear vários tipos de números – incluindo binários, decimais, hexadecimais, com ou sem sinal e também números decimais codificados em binários e números decimais com ponto flutuante. Analisaremos cada um deles neste capítulo.

Para compreender como seu computador trabalha com números, é essencial um bom conhecimento do indicador mais atarefado do microprocessador 6502B/65C02: o indicador de transporte do registro de status, discutido no Capítulo 3.

OBSERVANDO MELHOR O INDICADOR DE TRANSPORTE

A melhor maneira de obter um *zoom* do funcionamento do indicador de transporte (*carry flag*) é examiná-lo através de um "microscópio eletrônico" – ou seja, a nível de bits.

HEXADECIMAL	BINÁRIO
$04	0100
+ $01	+ 0001
$05	0101
$08	1000
+ $03	+ 0011
$0B	1011

Figura 10.1 Somando, sem transporte, nos sistemas hexadecimal e binário.

A Figura 10.1 compara dois problemas de adição com 4 bits, um executado com números hexadecimais e outro com binários. Podemos observar que nenhuma das operações resulta em transporte.

A Figura 10.2 apresenta mais dois problemas de adição, com números maiores (de 8 bits). O primeiro problema não gera transporte, mas o segundo sim. A soma, no segundo problema, é um número de 9 bits – 1 1000 1100 em binário ou 18C em hexadecimal.

HEXADECIMAL	BINARY
$8E	1000 1110
+ $23	+ 0010 0011
$B1	1011 0001
$8D	1000 1101
$FF	1111 1111
$018C	(1) 1000 1100

Figura 10.2 Mais dois problemas de adição em hexa e binário.

O Programa 10.1 é um programa em linguagem assembly que executa a segunda adição da Figura 10.1

Programa 10.1
ADDNCARRY
Adição em 8 bits com transporte

```
 1 *
 2 * ADDNCARRY
 3 *
 4  ORG$8000
 5 *
 6  CLD
 7  CLC
 8  LDA #$8D
 9  ADC #$FF
10  STA $FB
11  RTS
```


Digite o Programa 10.1, compile-o e, então, rode-o, usando o monitor de linguagem de máquina de seu Apple. Verifique o conteúdo do endereço de memória $FB: simplesmente digite o endereço FB e verá uma linha como esta:

```
00FB-  8C
```

Esta linha lhe mostrará que o endereço $FB agora contém o número $8C. Isto não é a soma dos números $8D e $FF, mas quase. Em aritmética hexadecimal, a soma de $8D e $FF é $18C – exatamente a soma que obtivemos, mais um transporte . E onde está esse transporte? Ele tem que ter sido colocado no bit de transporte do registro de status do processador.

PROCURANDO UM BIT EM UM PALHEIRO

Procurar um bit de transporte dentro de um Apple pode parecer a procura de uma agulha num palheiro, mas não é tão difícil, se soubermos para onde olhar. Uma forma de localizar o transporte que falta no programa ADDNCARRY é obtida através da inserção de algumas linhas adicionais no programa. O Programa 10.2 é essa versão aperfeiçoada, com as linhas extra.

Programa 10.2
ADDNCARRY2
Adição com Transporte *(Versão Aperfeiçoada)*

```
 1 *
 2 * ADDNCARRY2
 3 *
 4  ORG $8000
 5 *
 6  CLD
 7  CLC
 8  LDA #$8D
 9  ADC #$FF
10  STA $FB
11  LDA #0
12  ROL A
13  STA $FC
14  RTS
```

Nas linhas agora incluídas o acumulador é limpo e o operador de deslocamento de bits, ROL, é então usado para rodar o bit de transporte do registro P para o

acumulador. A seguir, o conteúdo do acumulador é depositado no registro de memória $FC, com uma instrução STA. Se esta rotina funcionar, encontramos nosso bit perdido...

Para verificar se o programa funciona, digite-o, compile-o e faça-o rodar. A seguir, observe os endereços de memória $FB e $FC, usando o monitor de linguagem de máquina, e veja se o cálculo efetuado pelo programa ADDNCARRY resultou num transporte. Eis, no formato do monitor do Apple II, o que aqueles dois registros deverão conter:

```
00FB- 8C 01
```

Esta linha nos diz que o endereço $FB novamente contém o número $8C (o resultado do cálculo ADDNCARRY, sem o transporte) e que o transporte resultante, reside, agora, no registro de memória $00FC.

UM PROGRAMA DE ADIÇÃO EM 16 BITS

O Programa 10.3, ADD16, realiza adições com números de 16 bits. O mesmo princípio ali adotado pode ser usado para escrever programas que adicionem números de 24, 32 ou mais bits.

Programa 10.3
ADD16
Um Programa de Adição em 16 bits

```
 1  *
 2  * ADD16
 3  *
 4  * ESTE PROGRAMA SOMA O NUMERO DE 16 BITS EM $FB E $FC
 5  * COM O NUMERO DE 16 BITS EM $FD E $FE
 6  * E DEPOSITA O RESULTADO EM $0300 E $0301
 7  *
 8  ORG $8000
 9  *
10  CLD
11  CLC
12  LDA $FB ; REM METADE INFERIOR DO NUMERO DE 16 BITS EM $FB E $FC
13  ADC $FD ; REM METADE INFERIOR DO NUMERO DE 16 BITS EM $FD E $FE
14  STA $0300 ; BYTE INFERIOR DA SOMA
15  LDA $FC ; REM METADE SUPERIOR DO NUMERO DE 16 BITS EM $FB E $FC
16  ADC $FE ; REM METADE SUPERIOR DO NUMERO DE 16 BITS EM $FD E $FE
17  STA $0301 ; BYTE SUPERIOR DA SOMA
18  RTS
```


Ao observar este programa, lembre que seu computador Apple armazena números de 16 bits com o byte de ordem inferior, em primeiro lugar, e o byte de ordem superior, em segundo lugar – o contrário do que se poderia esperar. Compreendida esta característica, comum a todos os computadores baseados no processador 6502/6502B/65C02, não é difícil entender a adição binária em 16 bits.

Neste programa, primeiramente desativamos o indicador de transporte do registro P. A seguir, adicionamos o byte inferior do número de 16 bits armazenado em $FB e $FC ao byte inferior do número de 16 bits armazenado em $FD e $FE.

O resultado desta metade do nosso cálculo é, então, colocado no endereço de memória $0300. Se existir um transporte, o carry bit do registro P será ativado automaticamente.

Na segunda metade do programa, o byte superior do número em $FB e $FC é somado com o byte superior do número em $FD e $FE. Se o bit de transporte do registro P foi ativado pelo resultado da operação de adição precedente, um transporte será também adicionado aos bytes superiores dos dois números. Se o bit de transporte estiver desativado, não há transporte.

Quando esta metade do nosso cálculo for completada, seu resultado é depositado no endereço de memória $0301. Finalmente, o resultado completo da adição é armazenado (byte inferior primeiro!) nos endereços $0300 e $0301.

SUBTRAÇÃO EM 16 BITS

O Programa 10.4 é um programa de subtração em 16 bits.

Programa 10.4
SUB16
Um Programa de Subtração em 16 Bits

```
 1  *
 2  * SUB16
 3  *
 4  * ESTE PROGRAMA SUBTRAI UM NUMERO DE 16 BITS EM $FB E $FC
 5  * DE UM NUMERO DE 16 BITS EM $FD E $FE
 6  * E DEPOSITA O RESULTADO EM $0300 E $0301
 7  *
 8  ORG $8000
 9  *
10  CLD
11  SEC ; REM ATIVAR TRANSPORTE
```

```
12  LDA $FD ; REM METADE INFERIOR DO NUMERO DE 16 BITS EM $FD E $FE
13  SBC $FB ; REM METADE INFERIOR DO NUMERO DE 16 BITS EM $FB E $FC
14  STA $0300 ; BYTE INFERIOR DO RESULTADO
15  LDA $FE ; REM METADE SUPERIOR DO NUMERO DE 16 BITS EM $FD E $FE
16  SBC $FC , REM METADE SUPERIOR DO NUMERO DE 16 BITS EM $FB E $FC
17  STA $0301 ; BYTE SUPERIOR DO RESULTADO
18  RTS
```

Como a subtação é exatamente o oposto da adição, o indicador de transporte é ativado, em lugar de ser desativado, antes que se realize uma operação de subtração em aritmética binária 6502B/65C02. Na subtração, o indicador de transporte é tratado como um empréstimo, não como um transporte, e deve, então, ser ativado, de forma que, se for necessário um empréstimo, exista um valor do qual emprestar.

Após o ativamento do bit de transporte, a operação de subtração é bastante simples; em nosso problema, o número de 16 bits armazenado em $FB e $FC é subtraído, byte inferior em primeiro lugar, do número de 16 bits armazenado em $FD e $FE. O resultado do problema – incluindo, se necessário, um empréstimo do byte superior – é então armazenado nos endereços $0300 e $0301, byte inferior primeiro.

MULTIPLICAÇÃO BINÁRIA

Os números binários são multiplicados da mesma forma que os números decimais. A Figura 10.3 é um exemplo de multiplicação binária. Infelizmente, não existem instruções em linguagem assembly 6502B/65C02 para multiplicação ou divisão. Para multiplicar dois números em linguagem assembly 6502B/65C02 é necessário realizar uma série de operações de adição; para efetuar uma divisão, uma série de operações de subtração.

```
    0110      ($06)
  × 0101      ($05)
  ------
    0110
   0000
  0110
 0000
 -------
 0011110      ($1E)
```

Figura 10.3 Um exemplo de multiplicação binária.

Observando cuidadosamente o problema de multiplicação da Figura 10.3, você notará que não é difícil desmembrar um problema de multiplicação em uma série de problemas de adição.

No exemplo, o número binário 0110 é, primeiramente, multiplicado por 1, e o resultado – também 0110, é claro – é escrito.

A seguir, 0110 é multiplicado por zero, e o resultado – uma linha de zeros – é deslocado um casa para a esquerda, ao ser escrito. Continuando, 0110 é multiplicado novamente por 1, e o resultado novamente deslocado para a esquerda. Finalmente, outra multiplicação por zero resulta em nova série de zeros, também escrita uma casa para a esquerda.

Por fim, todos os produtos parciais são somados, da mesma forma que o seriam em um problema de multiplicação convencional. O resultado é o produto final $1E.

Esta técnica de multiplicação funciona bem, mas é apenas um método arbitrário. Por que, por exemplo, deslocamos cada produto parcial para a esquerda, ao escrevê-lo? Teríamos obtido o mesmo resultado se houvéssemos deslocado os produtos parciais *para a direita* antes de somá-los.

Na multiplicação 6502B/65C02, isto é exatamente o que se costuma fazer: em lugar de deslocar cada produto parcial para a esquerda, antes de armazená-lo na memória, muitos dos algoritmos de multiplicação 6502B/65C02 deslocam o produto parcial precedente para a direita, antes de somá-lo ao novo.

O Programa 10.5 apresenta este método.

Programa 10.5
MULT16
Um Programa de Multiplicação em 16 Bits

```
1   *
2   * MULT16
4   MPR EQU $FD ; MULTIPLICADOR
5   MPD1 EQU $FE ; MULTIPLICANDO
6   MPD2 EQU $0300 ; NOVO MULTIPLICANDO APOS 8 DESLOCAMENTOS
7   PRODL EQU $0301 ; BYTE INFERIOR DO PRODUTO
8   PRODH EQU $0302 ; BYTE SUPERIOR DO PRODUTO
9   *
10  ORG $8000
11  *
12  * ESTES SAO OS NUMEROS QUE IREMOS MULTIPLICAR
13  *
```

```
14      LDA#250
15      STA MPR
16      LDA#2
17      STA MPD1
18    *
19    MULT CLD
20      CLC
21      LDA#0 ; LIMPAR ACUMULADOR
22      STA MPD2 ; LIMPAR ENDEREÇO PARA O MULTIPLICANDO DESLOCADO
23      STA PRODH ; LIMPAR BYTE SUPERIOR DO ENDERECO DO PRODUTO
24      STA PRODL ; LIMPAR BYTE INFERIOR DO ENDERECO DO PRODUTO
25      LDX#8 ; USAREMOS O REGISTRO X COMO UM CONTADOR
26    LOOP LSR MPR ; DESLOCAR MULTIPLICADOR PARA A DIREITA ; LSB CAI PARA O BIT DE
TRANSPORTE
27      BCC NOADD ; VERIFICAR BIT DE TRANSPORTE; SE ZERO, DESVIAR PARA NOADD
28      LDA PRODH
29      CLC
30      ADC MPD1 ; ADICIONAR BYTE SUPERIOR DO PRODUTO AO MULTIPLICANDO
31      STA PRODH ; O RESULTADO E O NOVO BYTE SUPERIOR DO PRODUTO
32      LDA PRODL ; CARREGAR ACUMULADOR COM O BYTE INFERIOR DO PRODUTO
33      ADC MPD2 ; ADICIONAR A PARTE ALTA DO MULTIPLICANDO
34      STA PRODL ; O RESULTADO E O NOVO BYTE INFERIOR DO PRODUTO
35    NOADD ASL MPD1 ; DESLOCAR MULTIPLICANDO PARA A ESQUERDA; BIT 7 CAI PARA O
TRANSPORTE
36      ROL MPD2 ; RODAR O BIT DE TRANSPORTE PARA O BIT 7 DE MPD1
37      DEX ; DECREMENTAR O CONTEUDO DO REGISTRO X
38      BNE LOOP ; SE O RESULTADO E DIFERENTE DE ZERO, VOLTAR AO LOOP
39      RTS
```

Obs: LSB = bit menos significativo

Como vimos, a multiplicação binária em 8 bits não é algo instantâneo. Existem muitos deslocamentos de bits, para a esquerda e para a direita, e é difícil acompanhá-los. No Programa 10.5, a manipulação mais difícil de se acompanhar é, provavelmente, a que envolve o multiplicando (MPD1 e MPD2). O multiplicando é apenas um valor de 8 bits, mas é tratado como um valor de 16 bits, por ir sendo deslocado para a esquerda; conforme se movimenta, ele necessita de um endereço de 16 bits (na realidade, dois endereços de 8 bits) para acomodá-lo.

Para observar como o programa funciona, digite-o e compile-o, utilizando a seguir o comando G de seu monitor para executá-lo. Mantendo-se no modo monitor, consulte os conteúdos dos endereços $0301 e $0302. Estes dois registros devem, agora, conter o número $01F4 (byte inferior primeiro), que é o equivalente hexadecimal do número decimal 500. Este, é claro, é o produto dos números decimais 2 e 250 – a multiplicação efetuada pelo programa.

UM PROGRAMA DE MULTIPLICAÇÃO APERFEIÇOADO

Embora o Programa 10.5 funcione bem, ele não é o único programa de multiplicação em 16 bits disponível; de fato, ele até não é dos melhores. Existem muitos algoritmos para multiplicação binária, sendo alguns menores e mais eficientes. O Programa 10.6, por exemplo, é consideravelmente menor, sendo portanto, mais rápido e mais eficiente, em termos de memória.

Uma das melhores características deste programa aperfeiçoado é a utilização do acumulador do 6502B/65C02, em lugar de um endereço de memória, para armazenamento temporário dos resultados do problema.

Programa 10.6
MULT16B
Um Programa Aperfeiçoado de Multiplicação em 16 Bits

```
 1 *
 2 * MULT16B
 3 * (UM PROGRAMA APERFEIÇOADO DE MULTIPLICACAO EM 16 BITS)
 4 *
 5 PRODL  EQU $FD
 6 PRODH  EQU $FE
 7 MPR    EQU $0300
 8 MPD    EQU $0301
 9 *
10  ORG $8000
11 *
12 VALUES LDA #10
13  STA MPR
14  LDA #10
15  STA MPD
16 *
17  LDA #0
18  STA PRODH
19  LDX #8
20 LOOP LSR MPR
21  BCC NOADD
22  CLC
23  ADC MPD
```

Da mesma forma usada para testar o Programa 10.5, execute e verifique os resultados do Programa 10.6, utilizando o monitor de linguagem de máquina.

Você pode efetuar experiências com estes dois problemas de multiplicação, tentando valores diferentes e verificando como eles são processados nos dois programas.

A melhor maneira de familiarizar-se com a multiplicação binária, entretanto, é resolver alguns problemas com lápis e papel, o que o levará rapidamente à compreensão dos princípios da multiplicação 6502B/65C02.

DIVISÃO BINÁRIA DE MULTIPRECISÃO

No decorrer deste capítulo, demonstramos que a adição é o inverso da subtração. Da mesma forma, a divisão não é nada além do inverso da multiplicação. Já sabemos também que o microprocessador 6502B/65C02, que não tem instruções específicas para a multiplicação, também não as apresenta quanto à divisão.

Mas é possível realizar divisões – mesmo longas divisões de multiprecisão – se, da mesma forma que os problemas de multiplicação eram desmembrados em seqüências de adições, desmembrarmos os problemas de divisão em seqüência de subtrações.

O Programa 10.7, por exemplo, é uma rotina projetada para dividir um número por outro, desmembrando o processo de divisão numa série de rotinas de subtração. Durante a execução do programa, a parte superior do dividendo será armazenada no acumulador, e a parte inferior na variável chamada DVDL.

O programa contém um bom número de deslocamentos, rotações, subtrações e decrementos do registro X. Quando o corpo principal do programa termina, o quociente é armazenado em uma variável, QUOT, e o resto permanece no acumulador. Na linha 38, o resto é movido do acumulador para uma variável chamada RMDR. Uma instrução RTS encerra o programa.

Programa 10.7
DIV 8.16
Um Programa de Divisão Binária

```
1     *
2     * DIV8.16
3     *
4     ORG $8000
5     *
6     DVDH EQU $FD ; PARTE INFERIOR DO DIVIDENDO
7     DVDL EQU $FE ; PARTE SUPERIOR DO DIVIDENDO
8     QUOT EQU $0300 ; QUOCIENTE
9     DIVS EQU $0301 ; DIVISOR
10    RMDR EQU $0302 ; RESTO
11    *
```



```
12    LDA# $1C ; SIMPLESMENTE UM VALOR-EXEMPLO
13    STA DVDL
14    LDA#$02 ; O DIVIDENDO E AGORA $021C
15    STA DVDH
16    LDA#$05 ; OUTRO VALOR-EXEMPLO
17    STA DIVS ; ESTAMOS DIVIDINDO POR 5
18    *
19    LDA DVDH ; O ACUMULADOR CONTERA DVDH
20    LDX#08 ; PARA UM DIVISOR DE 8 BITS
21    SEC
22    SBC DIVS
23  DLOOP PHP ; SALVAR O REGISTRO P (ROL E ASL AFETAM-NO)
24    ROL QUOT
25    ASL DVDL
26    ROL A
27    PLP ; RESTAURAR REGISTRO P
28    BCC ADDIT
29    SBC DIVS
30    JMP NEXT
31  ADDIT ADC DIVS
32  NEXT DEX
33    BNE DLOOP
34    BCS FINI
35    ADC DIVS
36    CLS
37  FINI ROL QUOT
38    STA RMD
39    RTS
```

RODANDO O PROGRAMA

O Programa 10.7 pode ser usado para dividir qualquer número de 16 bits, sem sinal, por qualquer número de 8 bits, também sem sinal. Na forma em que está escrito, ele divide o número hexadecimal $021C (540 em notação decimal) por 5. O quociente é armazenado no registro de memória $0300 e o resto, se existir, é armazenado no registro de memória $0302.

Digite o programa, compile-o e execute-o, e use seu monitor para inspecionar o conteúdo dos endereços $0300 e $0302. O endereço $0300 deverá conter o número hexadecimal $6C (decimal 108), e deverá haver um zero no endereço $0302, já que o quociente de 540 dividido por 8 é 108, e a divisão é exata.

ULTIMATUM...

Como se pode ver, é ainda mais difícil escrever uma rotina de divisão, para o Apple, que um programa de multiplicação. De fato, a escrita de quase todos os programas de multiprecisão matemática, para um computador de 8 bits, é muito trabalhosa. Quando você precisar escrever um programa no qual apenas alguns cálculos devam ser realizados, poderá, por vezes, usar rotinas simples como as apresentadas neste capítulo.

Mas, a linguagem assembly não é, usualmente, a melhor para a escrita de programas longos e complexos que envolvam muita matemática de múltipla precisão. Caso precise fazê-lo, você poderá julgar de grande valia escrever parte do programa em assembly e parte em BASIC – a parte que envolve toda a matemática.

Desta forma, poderá tirar proveito da excelente matemática de ponto flutuante que é residente no interpretador BASIC do Apple. Poderá ser, inclusive, mais interessante escrever o programa em BASIC, Pascal, COBOL, Logo, ou outras linguagens de alto nível, a fazê-lo em linguagem assembly.

Se, apesar desta advertência, você ainda desejar escrever rotinas matemáticas complexas em assembly 6502B/65C02, existem alguns livros que poderão auxiliá-lo, como o *6502 Assembly Language Subroutines*, de Lance A. Leventhal e Winthrop Saville, publicado pela Osborne/McGraw-Hill. Existem, também, várias rotinas matemáticas do tipo "digite-e-rode" em alguns dos manuais e textos citados na Bibliografia deste livro.

NÚMEROS COM SINAL

Para representar um número com sinal em aritmética binária, tudo o que temos a fazer é deixar que o bit mais à esquerda (bit 7) represente um sinal positivo ou negativo. Em aritmética binária, se o bit 7 de um número é 0, o número é positivo; se é 1, o número é negativo.

Obviamente, se usamos um bit de um número de 8 bits para representar seu sinal, o número deixa de ser um número de 8 bits. O que teremos é um número de 7 bits, ou – se expressarmos de outra maneira – teremos um número que pode representar valores de –128 a + 127, em lugar de 0 a 255.

ADIÇÃO BINÁRIA COM SIGNAL

É necessário mais do que a redesignação de um bit para fazer com que as operações aritméticas binárias convertam-se em operações com sinal. Consideremos, por exemplo, o que obteríamos se tentássemos somar os números +5 e –4, nada mais executando além de usar o bit 7 como sinal:

```
  0000 0101  (+5)
+ 1000 0100  (–4)
-------------------
  1000 1001  (–9)
```

A resposta está errada; deveríamos obter + 1.

Isto ocorreu porque tentamos solucionar o problema sem utilizar um conceito que é fundamental na aritmética binária com sinal: o conceito de *complementos*.

Os complementos são usados em aritmética binária com sinal pelo fato dos números negativos serem complementares dos positivos, e dos complementos serem muito fáceis de se calcular em aritmética binária: o complemento de 0 é 1, e o de 1 é 0.

PRIMEIRO E SEGUNDO COMPLEMENTOS

É razoável assumir que se pode chegar ao complemento negativo de um número binário positivo complementando cada 0 do número para 1 e cada 1 para 0 (exceto o bit 7, é lógico, que deverá ser usado para representar o sinal).

Esta técnica de cálculo do complemento de um número pela inversão dos bits 0 para 1 e 1 para 0 tem um nome, em linguagem assembly: é *primeiro complemento*.

Para verificar se a técnica do primeiro complemento funciona, vamos tentar usá-la para somar +8 com –5:

```
  0000 1000  (+8)
+ 1111 1010  (–5)  (primeiro complemento)
-------------------
  0000 0010  (+2)  (mais transporte)
```

Errado, novamente! A resposta deveria ser +3. Isto nos leva de volta à prancheta de desenho... o primeiro complemento não funciona.

Felizmente, existe uma outra técnica, e esta dá certo. É a técnica do *segundo*

complemento. Para usá-la, primeiramente calculamos o primeiro complemento de um número positivo. A seguir, adicionamos 1. Isto nos dá o segundo complemento – o complemento real – do número. E, então, podemos usar as regras convencionais da matemática binária para números com sinais.

Eis dois exemplos de adição com o segundo complemento:

```
0000 0101   (+5)
1111 1000   (–8)   (segundo complemento)
---------
1111 1101   (–3)

1111 1011   (–5)   (segundo complemento)
0000 1000   (+8)
---------
0000 0011   (+3)   (mais transporte)
```

ALGUNS EXEMPLOS

Não é fácil explicar porque o método do segundo complemento funciona; mas, após trabalhar um pouco com números binários com sinal, você perceberá. Ajuda lembrar que o bit mais alto de um número binário é sempre interpretado como o sinal, em notação de segundo complemento, de forma que um número com o bit mais alto ativado é sempre interpretado como um número negativo. Assim, o número hexadecimal $7F, que equivale ao decimal 127, é o maior número positivo que pode ser expresso em notação de segundo complemento em 8 bits.

Incrementando o número hexa $7F, vemos que isto é verdade. Em notação binária, $7F é escrito como %0111 1111 (um número binário no qual o bit superior não está ativado). Incrementando $7F, obtemos $80, ou %1000 0000, um número que tem o bit superior ativado, e será então interpretado em notação de segundo complemento em 8 bits como –128, e não como +128. Portanto, o maior número positivo que pode ser expresso naquela notação é 127.

Vejamos, agora, alguns números binários negativos. Em aritmética do segundo complemento, os números negativos começam em –1 e prosseguem para trás, como os números na aritmética convencional. Nesta, não existe o número –0; assim, ao decrementar um 0, o que obtemos não é "menos 0", mas –1. Decrementando –1 temos –2, que pode parecer um número maior, mas, na realidade, é menor. (Se, por exemplo, decrementarmos –2, obteremos – 3.)

A aritmética de segundo complemento trabalha de forma similar. Se decrementarmos 0 usando aritmética de segundo complemento em 8 bits, obteremos $FF, que equivale a – 1 em notação decimal. Decrementando $FF em segundo complemento, obteremos $FE, o binário de 8 bits com sinal equivalente a –2. O número decimal –3 é escrito como $FD em notação de segundo complemento, e o decimal –4 é escrito como $FC.

Trabalhando nessa direção, eventualmente descobriremos que o menor número negativo que pode ser expresso em notação de segundo complemento em 8 bits é o número hexadecimal $80, que equivale a –128 em numeração decimal.

Para começar a escrever programas que usam números binários com sinal, é necessário, na realidade, instruções bem mais detalhadas. Este capítulo tem somente a finalidade de apresentar algumas das técnicas usadas em programas que contêm números binários com sinal.

USANDO O INDICADOR DE "OVERFLOW"

Em aritmética binária com sinal, ao realizar cálculos usando números com sinal, o indicador de "overflow" (V) do registro de *status* do processador – e não o indicador de transporte – é usado para levar números de um byte a outro.

A razão disto é que o indicador de transporte é ativado quando há um "overflow" no bit 7 de um número binário; porém, quando se trata de um número com sinal, o bit 7 é o bit *do sinal* e não faz parte do número. Assim, o indicador de transporte não pode ser usado para detectar um transporte em operações que envolvam números com sinal.

Podemos, entretanto, usar o bit de "overflow" do registro de *status* do processador. Este bit é ativado quando há um transbordamento no bit 6, e pode, portanto, ser usado como um bit de transporte em operações aritméticas com números com sinal.

Como você deve se lembrar da álgebra do colegial, as regras de adição, subtração, multiplicação e divisão são um tanto complexas: elas variam, de acordo com os sinais dos números envolvidos no cálculo e com o tipo de cálculo realizado.

Não deve causar surpresa, então, o fato de que as regras para uso do indicador de "overflow" em cálculos que envolvem números binários com sinal sejam também complicadas. Você poderá encontrá-las em livros sobre programação avançada em linguagem assembly; elas estão muito além do escopo deste capítulo.

NÚMEROS BCD (DECIMAIS CODIFICADOS EM BINÁRIO)

Em notação BCD, os dígitos de 0 a 9 são expressos da mesma forma que na notação binária convencional, mas os dígitos hexadecimais de $A a $F (%1010 a %1111 em binário) não são usados. Números longos, portanto, têm de ser representados de maneira diferente em notação BCD.

O número decimal 1258, por exemplo, seria escrito em notação BCD como

1	2	5	8
0000 0001	0000 0010	0000 0101	0000 1000

Em notação binária convencional, o mesmo número seria escrito como

$0	$4	$E	$A
0000	0100	1110	1010

ou seja, $04EA, o equivalente hexadecimal de 1258.

A notação BCD é, geralmente, usada em programas de contabilidade, uma vez que, ao contrário da aritmética binária direta, a BCD apresenta sempre resultados 100% precisos.

Os números BCD são, também, usados quando se deseja que sejam apresentados instantaneamente, dígito por dígito, conforme vão sendo usados (por exemplo, para apresentação constante do placar, em programas de jogos).

A principal desvantagem dos números BCD é a dificuldade de utilização: ao usar números BCD, é preciso ser extremamente cuidadoso com os sinais, pontos decimais e operações de transporte. É preciso também decidir se se deseja usar um byte de 8 bits para cada dígito (o que desperdiça memória, já que são necessários apenas 4 bits para codificar um dígito BCD) ou se se pefere comprimir dois dígitos por byte, o que economiza memória, mas consome tempo de processamento.

NÚMEROS EM PONTO FLUTUANTE

Números em ponto flutuante são os que permitem aos computadores e calculadoras realizarem cálculos com valores decimais e frações. A maioria das calculadoras usa números em ponto flutuante para realização dos cálculos matemáticos, como também o interpretador BASIC residente em seu Apple.

No sistema Apple de ponto flutuante, os números são desmembrados em três partes: um expoente, uma mantissa e um sinal. Estas partes são armazenadas em um bloco de memória chamado acumulador de ponto flutuante, que reside nos registros de memória de $61 a $66. Existe ainda outro acumulador de ponto flutuante, de $69 a $6E.

Infelizmente, para os programadores em linguagem assembly, é extremamente difícil compreender como funcionam as rotinas em ponto flutuante, e ainda mais difícil escrevê-las. Porém, lembremos que o Apple possui um ótimo sistema residente de ponto flutuante. Para usá-lo em um programa em assembly, tudo o que se tem a fazer é escrever o programa parcialmente em linguagem assembly e parcialmente em BASIC, interconectando estas partes através do uso da função USR(X), explicada no Capítulo 5.

Ao escrever desta forma um programa, pode-se usar linguagem assembly para as secções do programa que requerem alta velocidade ou alto desempenho – e, para as porções do programa que necessitam matemática de alta precisão, pode-se usar BASIC para acessar o sistema residente de ponto flutuante. (É possível acessar, também, as rotinas de ponto flutuante a partir de programas em assembly, desde que se saiba como converter números em ponto flutuante para números binários e vice-versa, mas este processo está um pouco além do escopo deste capítulo.)

Uma vez que o sistema residente de ponto flutuante existe e é fácil de usar, você poderá prescindir do conhecimento da maioria das técnicas de programação descritas neste capítulo. Porém, uma compreensão do funcionamento delas irá torná-lo, sem dúvida, um melhor programador em assembly.

Além disso, você deverá conhecer, pelo menos, os fundamentos da aritmética 6502B/65C02, se deseja tornar-se um bom programador em linguagem assembly Apple. Afinal, processamento matemático, de uma forma ou de outra, é tudo o que a programação de computadores é, na realidade.

Uma vez que seu Apple soma, subtrai, compara e desloca bits ao longo de qualquer programa que processa, seria difícil escrever programas sem, ao menos, conhecer algo sobre operações aritméticas com números binários.

Assim, mesmo que você nunca precise escrever uma rotina em assembly para realizar longas divisões em números com sinal, com precisão de 17 casas decimais, há boas chances de que você necessite, pelo menos, usar algumas operações aritméticas em alguns dos programas que escrever.

Portanto, antes de passar para o próximo, certifique-se de haver compreendido perfeitamente este capítulo.

CAPÍTULO 11

A MÁGICA DA MEMÓRIA

Os engenheiros que projetaram o Apple IIc e o Apple IIe conseguiram realizar uma verdadeira façanha: colocaram mais de 128K de memória em máquinas de 64K. Este segredo pode ser explicado em duas palavras: chaveamento de bancos.

O chaveamento de bancos, como vimos no Capítulo 3, é baseado no conceito de que dois blocos de memória podem compartilhar o mesmo espaço de endereçamento, desde que não tentem ocupar esse espaço ao mesmo tempo.

No caso da adoção do sistema de chaveamento de bancos em um computador, para vários blocos de RAM e ROM são assinalados endereços idênticos, sendo então incorporadas características especiais de chaveamento que permitem selecionar um ou outro dos blocos de memória que compartilham aquele mesmo espaço de endereçamento.

Os programadores podem, então, movimentar blocos específicos de memória de e para o espaço disponível, de acordo com as necessidades do programa.

Nos computadores Apple, o chaveamento de bancos é realizado com o auxílio de circuitos eletrônicos especiais, chamados *soft switches*. São circuitos que podem ser usados exatamente como chaves (comutadores) controladas por softwares podendo ser usadas, em programas, para definir quais blocos da secção chaveada da memória ocuparão determinados endereços. Os *soft switches* podem também ser usados para proteger certos blocos de memória, tornando impossível gravar nesses blocos, mas permitindo que sejam lidos.

MEMÓRIA: ALGUNS CONCEITOS BÁSICOS

Os conceitos básicos de gerenciamento de memória auxiliam a compreensão do sistema de chaveamento de bancos. Eis alguns dos princípios de mapeamento de memória em microcomputadores.

QUE DIFERENÇA FAZ UM "K"

A letra K, abreviatura de *kilobyte*, representa o número decimal 1024. Este equivale ao hexadecimal $400. O nome *kilobyte* decorre do fato de ser muito próximo ao valor decimal 1000, abreviado como *kilo* ou simplesmente *K*. Assim, um computador de 64K pode endereçar 65536 (1024x64) bytes de memória. Um computador de 128K apresenta o dobro dessa capacidade: 131072 (1024x128) bytes de memória.

O CONCEITO DE "PÁGINA"

No mundo dos computadores de 8 bits, um bloco de 256 endereços de memória (usualmente numerados de 0 a 255) é chamado de *página*. Uma página é uma unidade de medida muito conveniente quando se trata de microcomputadores, uma vez que os números decimais de 0 a 255 equivalem aos hexadecimais $00 a $FF, podendo, assim, ser expressos como os dois primeiros dígitos de um endereço hexadecimal de quatro dígitos.

Num Apple IIc ou IIe, a Página Zero (Zero Page) consiste dos endereços de memória de $00 a $FF, a Página Um inclui os endereços de $0100 a $01FF, e assim por diante. A página mais alta em um computador Apple Série II é a Página $FF, composta pelos endereços de $FF00 a $FFFF.

Em gráficos (e texto), a palavra "página" pode também referir-se a um segmento de memória usado para uma apresentação na tela. Este é um significado completamente diferente da palavra "página", com relação ao significado que ora consideramos. O contexto no qual a palavra é utilizada esclarece, normalmente, seu significado.

CHAVEAMENTO DE BANCOS

A Figura 11.1 é um mapa de memória que mostra como o chaveamento de bancos funciona num Apple IIc. Um Apple IIe com um cartão de 80 colunas expandido

apresenta, exatamente, o mesmo mapa de memória. Na Figura 11.1, existem duas faixas extensas, assinaladas como "RAM Principal" e "RAM Auxiliar". Estas faixas representam os 128 kilobytes de memória do Apple IIc, obtidos também com um Apple IIe totalmente expandido.

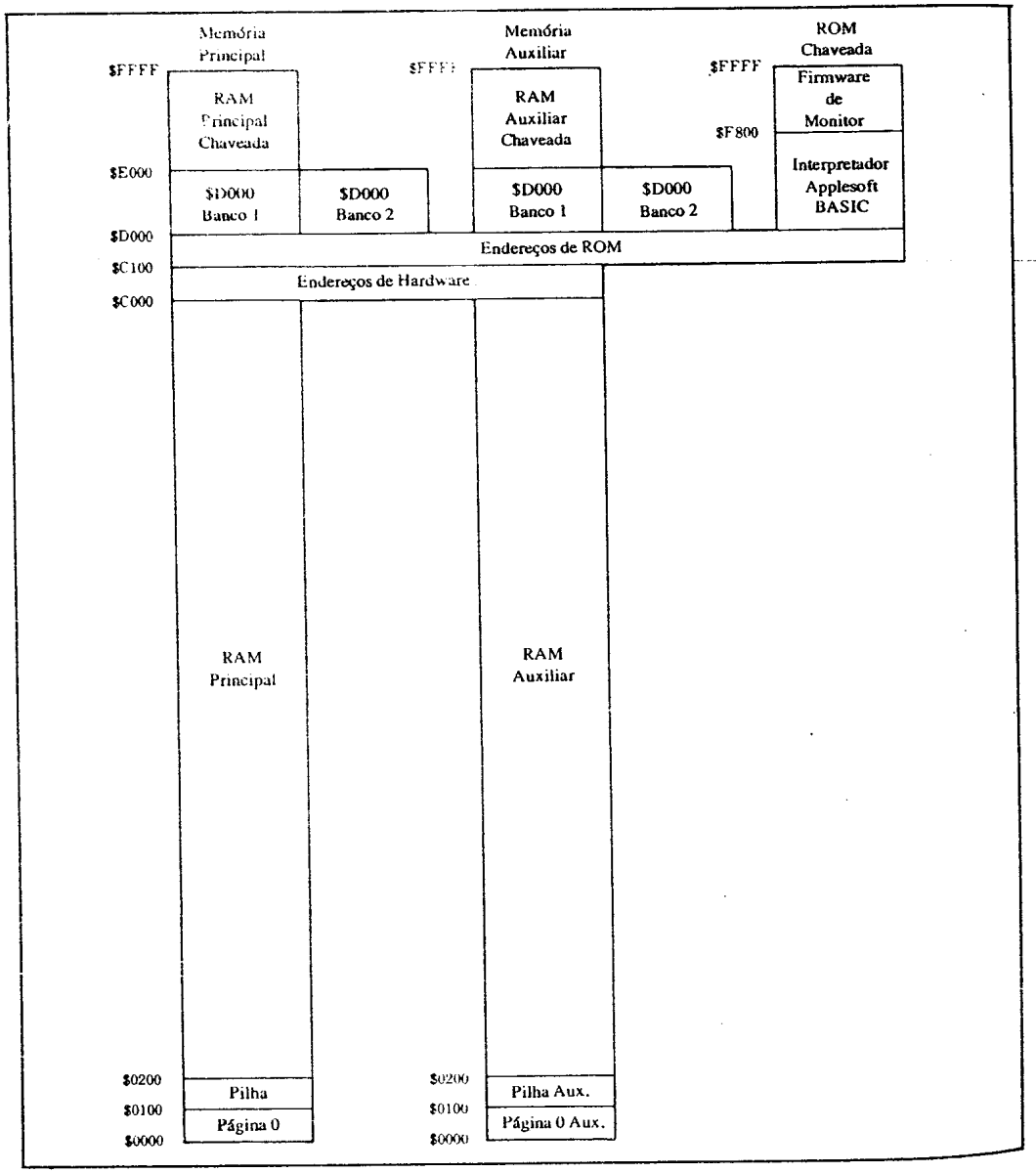


Figura 11.1 Chaveamento de bancos no Apple IIc e no Apple IIe totalmente expandido.

Examine os endereços que aparecem ao longo das colunas "Memória Principal" e "Memória Auxiliar" e verá que ambas as colunas utilizam a mesma série de endereços: o espaço de endereçamento de 64K que se estende de $0000 a $FFFF (ou de 0 a 65535 em notação decimal). À direita, existe uma coluna menor, intitulada "ROM Chaveada", que se estende entre os endereços $C100 e $FFFF.

Com o auxílio dos *soft switches*, é possível escrever programas em linguagem assembly que usam ambos os bancos, principal e auxiliar. Com os *soft switches*, quase todo o espaço de memória de 128K presente no mapa pode ser usado como RAM. Alternativamente, em programas que se utilizam do Applesoft BASIC ou do monitor residente, a ROM pode ser chaveada para o espaço de $D000 a $FFFF, permitindo que tanto o Applesoft BASIC como o monitor sejam usados nos programas.

RAM PRINCIPAL E AUXILIAR

O bloco de memória de 48K, de $0000 a $BFFF, pode ser usado como RAM principal ou como RAM auxiliar. Além disso, é possível escrever programas em assembly que leiam dados em um destes bancos e os gravam no outro. Desta forma, um programa armazenado em um espaço de memória pode ler e gravar dados em outro espaço.

Existem dois *soft switches* que podem ser usados para chavear entre os bancos de memória principal e auxiliar, em programas em linguagem assembly. Um deles, chamado RAMRD, é utilizado para selecionar memória principal ou memória auxiliar para leitura. O outro, RAMWRT, seleciona memória principal ou auxiliar para gravação.

O RAMRD ocupa três registros de memória: $C002, $C003 e $C013. O RAMWRT também ocupa três registros: $C004, $C005 e $C014.

Armazenando qualquer valor em $C002, *desativamos* o *soft switch* RAMRD e selecionamos *leitura* da memória *principal*. Armazenando qualquer valor em $C003, *ativamos* o *soft switch* RAMRD e selecionamos *leitura* da memória *auxiliar*.

Qualquer valor colocado em $C004 *desativa* o *soft switch* RAMWRT e seleciona memória *principal* para *gravação*. Qualquer valor em $C005 *ativa* RAMWRT e seleciona memória *auxiliar* para *gravação*.

Através da leitura dos valores presentes nos registros de memória $C013 e $C014, o programador pode verificar a condição dos *soft switches* RAMRD e RAMWRT. Se o registro $C013 apresenta o bit alto ativado, o RAMRD está ativado e, portanto, a memória auxiliar foi selecionada para leitura. Se o registro $C014 apresenta bit alto

ativado, o RAMWRT está ativado e, portanto, a memória auxiliar foi selecionada para gravação.

As funções de RAMRD e RAMWRT, bem como as de vários outros *soft switches* estão listadas na Tabela 11.3

MEMÓRIAS PRINCIPAL E AUXILIAR CHAVEADAS

Outro bloco de memória que pode ser controlado através de chaveamento é o segmento que se estende de $D000 a $FFFF. Ativando e desativando *soft switches* em várias combinações, um programador pode usar este banco de memória como memória principal, memória auxiliar ou ROM. Além disso, o bloco de memória de $D000 a $DFFF pode ser subdividido em dois: "$D000-Banco 1" e "$D000-Banco 2" na Figura 11.1. O bloco $D000 a $DFFF pode, portanto, ser usado de cinco maneiras diferentes: como Memória Principal – Banco 1, Memória Principal – Banco 2, Memória Auxiliar – Banco 1, Memória Auxiliar - Banco2 e ROM.

Todo o bloco compreendido entre $D000 e $FFFF pode ser controlado por uma série de *soft switches* localizados nos endereços de memória de $C080 a $C08F. Utilizando-os, é possível escrever programas em assembly que lerão a ROM localizada entre $D000 e $FFFF, enquanto gravam em RAM (tanto auxiliar quanto principal) residente no mesmo bloco de memória. A mesma série de *switches* também controla a seleção de $D000-Banco 1 ou $D000-Banco 2 para leitura e/ou gravação.

A Tabela 11.1 ilustra o uso dos *soft switches* de $C080 a $08F. Para usá-los, não é necessário colocar nenhum valor neles; por estranho que pareça, basta *ler* o *switch* para ativá-lo. Não é necessário efetuar nada com o valor lido – o próprio ato da leitura ativa o *switch*.

Eis um exemplo de como ativar o *switch* em $C080 com uma operação de leitura:

```
LDA $C080
```

O mnemônico LDA é, logicamente, uma instrução "de leitura". Em condições normais, a instrução LDA não afeta o conteúdo do registro de memória que a segue. No Apple IIc/IIe, entretanto, o endereço de memória $C080 é um dos *soft switches*, que são afetados por instruções "de leitura", como a instrução LDA. Assim, quando a instrução LDA $C080 é encontrada em um programa em linguagem assembly Apple IIc/IIe, o *soft*

switch em $C080 é ativado, e as instruções listadas para $C080 na Tabela 11.1 serão obedecidas.

Tabela **11.1** *Soft switches* de Seleção de Bancos

Endereço do Switch	Operação de dupla leitura?	Função
$C080 Banco 2	Não	Ler RAM; sem gravação; use $D000
$C081 Banco 2	Sim	Ler ROM; gravar em RAM; use $D000
$C082 Banco 2	Não	Ler ROM; sem gravação; use $D000
$C083 Banco 2	Sim	Ler e gravar RAM; use $D000
$C088 Banco 1	Não	Ler RAM; sem gravação; use $D000
$C089 Banco 1	Sim	Ler ROM; gravar em RAM; use $D000
$C08A Banco 1	Não	Ler ROM; sem gravação; use $D000
$C08B Banco 1	Sim	Ler e gravar RAM; use $D000

OPERAÇÕES DE DUPLA LEITURA

Quando um *switch* é tão sensível a ponto de poder ser ativado com o que, ordinariamente, é um comando somente de leitura, acidentes desagradáveis – se não desastrosos – podem ocorrer. Para evitar isto, os projetistas do Apple IIc/IIe incluíram proteções extras em alguns dos *soft switches* mais críticos das posições de memória de $C080 a $C08F. Para ativá-los, é necessário efetuar duas leituras – em outras palavras, duas operações "de leitura" consecutivas. Por exemplo, para ativar o *switch* na posição de memória $C083, temos de realizar um par de operações:

```
LDA $C083
LDA $C083
```

OUTROS "SOFT SWITCHES"

Além dos *switches* listados na Tabela 11.1, dois outros são, algumas vezes, utilizados em operações de chaveamento de bancos:

- O *switch* RDBNK2, localizado no endereço de memória $C011, que pode ser lido para determinar qual dos bancos está em uso: $D000, Banco 1, ou $D000, Banco2 . Se o bit 7 de RDBNK2 estiver ativado, o Banco 2 é o que está sendo usado; se estiver desativado, é o Banco 1.
- O *switch* RDLCRAM, localizado em $C012, que pode ser lido para determinar se a leitura está sendo efetuada em RAM ou ROM. Se o bit 7 de RDLCRAM estiver ativado, a RAM estará sendo lida; se estiver desativado, é a ROM.

DUAS PILHAS E DUAS PÁGINAS ZERO

Observando a parte inferior da Figura 11.1, entre as posições de memória $0000 e $01FF, veremos que existem duas pilhas e duas Páginas Zero – um par na memória principal e outro na auxiliar. Quando os *soft switches* RAMRD e RAMWRT são usados para determinar qual a memória – principal ou auxiliar – deve ser usada pelo programa, a pilha e a Página Zero que residem no banco de memória apropriado são automaticamente selecionadas.

Mas se não for desejável este processo automático de seleção, existe um *soft switch* que pode ser usado para permitir uma seleção manual da pilha e da Página Zero a serem usadas em um programa. Este *soft switch,* denominado ALTZP, consiste de três registros de memória: $C008, $C009 e $C016.

O armazenamento de qualquer valor em $C008 desativa o ALTZP, consiste de três registros de memória: $C008, $C009 e $C016.

O armazenamento de qualquer valor em $C008 desativa o ALTZP e seleciona a pilha e a Página Zero da memória principal, tanto para leitura quanto para gravação. O armazenamento de qualquer valor em $C009 ativa o *switch* e seleciona, para leitura e gravação, a pilha e a Página Zero da memória auxiliar.

O *soft switch* em $C016 permite verificar a condição do ALTZP. Se o bit superior de $C016 estiver ativado, indica que o ALTZP está ativado, o que corresponde à seleção da pilha e da Página Zero da memória alternativa. Se o bit superior de $C016 estiver desativado, o ALTZP está desativado, e a seleção corresponde à memória principal.

A utilização do ALTZP requer muito cuidado. Quando utilizado para mudar a localização da Página Zero, ele também muda a localização da pilha de hardware – e tal mudança, sem os devidos cuidados, pode arruinar o programa em andamento.

Outro problema em potencial é o fato do *switch* ALTZP ser sobrepujado pelo RAMRD e pelo RAMWRT, já que estes selecionam automaticamente a pilha e a Página Zero, a serem utilizadas no programa. Outra fonte de problemas é o sistema de gerenciamento de interrupções do Apple, que pode desativar o ALTZP sem aviso.

Antes de utilizar o ALTZP de forma segura, portanto, é necessário um melhor conhecimento sobre interrupções, operações com a pilha, uso dos *switches* RAMRD e RAMWRT, e algumas outras técnicas que são abordadas apenas superficialmente neste livro. Se você é um iniciante em programação assembly, deve considerar a utilização do *switch* ALTZP como algo a ser feito de forma bastante restrita.

MEMÓRIA NÃO PASSÍVEL DE CHAVEAMENTO

Os blocos de memória localizados entre $C000 e $CFFF são reservados para o sistema operacional do Apple IIc/IIe, não estando sujeitos a chaveamento. O conteúdo destes blocos de memória será abordado em detalhes na secção a seguir.

UM MAPA DE MEMÓRIA MAIS DETALHADO

A primeira secção deste capítulo representa apenas uma breve introdução ao gerenciamento de memória do Apple IIc/IIe. Iremos, agora, explorar esse mesmo assunto em maior profundidade, utilizando o mapa apresentado na Figura 11.2.

Ao analisar esse mapa, começaremos pela parte inferior – *memória inferior*, no jargão da computação – e trabalharemos no sentido ascendente, observando detalhada-

mente o conteúdo de alguns dos segmentos mais importantes da memória dos computadores Apple.

ENDEREÇOS DE $00 A $FF (PÁGINA ZERO)

A utilização da Página Zero permite acelerar a operação do programa. Quando um operando em linguagem de máquina pode ser expresso como um endereço da Página Zero, requer apenas um byte de memória, em lugar dos dois bytes requeridos quando o endereço não pertence à Pagina Zero. Portanto, o programa escrito com um menor número de bytes apresentará um processamento mais rápido.

Há, também, o fato muito importante de que alguns modos de endereçamento – especificamente os modos de endereçamento indireto – *requerem* o uso de endereços da Página Zero. Como os modos de endereçamento indireto são muito poderosos, a maioria dos programas em linguagem assembly utiliza extensivamente a Página Zero.

Infelizmente, nem sempre é fácil encontrar espaço livre na Página Zero para programas escritos pelo usuário. Isto se deve à Página Zero ser tão útil que os engenheiros que projetaram o computador utilizaram grande parte dela.

- O Applesoft BASIC usa os registros da Página Zero de $00 a $05, $0A a $18 e a maioria dos registros de $50 até $F8.
- O monitor de linguagem de máquina utiliza os registros da Página Zero de $00 a $19 e de $1E a $25.
- O sistema operacional de disco ProDOS utiliza os registros da Página Zero de $0A a $1E. Entretanto, o sistema ProDOS grava a informação nos registros de $10 a $1E antes de usá-la, e depois restaura o conteúdo desses registros. Efetivamente, portanto, os únicos registros que o ProDOS reclama para uso exclusivo são os de $0A a $0F.

Como a maior parte da Página Zero é usada pelo BASIC, monitor e ProDOS, existem apenas alguns pequenos blocos que permanecem livres para uso em programas do usuário, escritos sob ProDOS. A Tabela 11.2 apresenta a lista destas posições da Página Zero.

Se você utilizar unicamente as posições apresentadas na Tabela 11.2, não interferirá na operação do Applesoft BASIC, do ProDOS ou do monitor. No entanto, se precisar das outras posições da Página Zero, há várias opções:

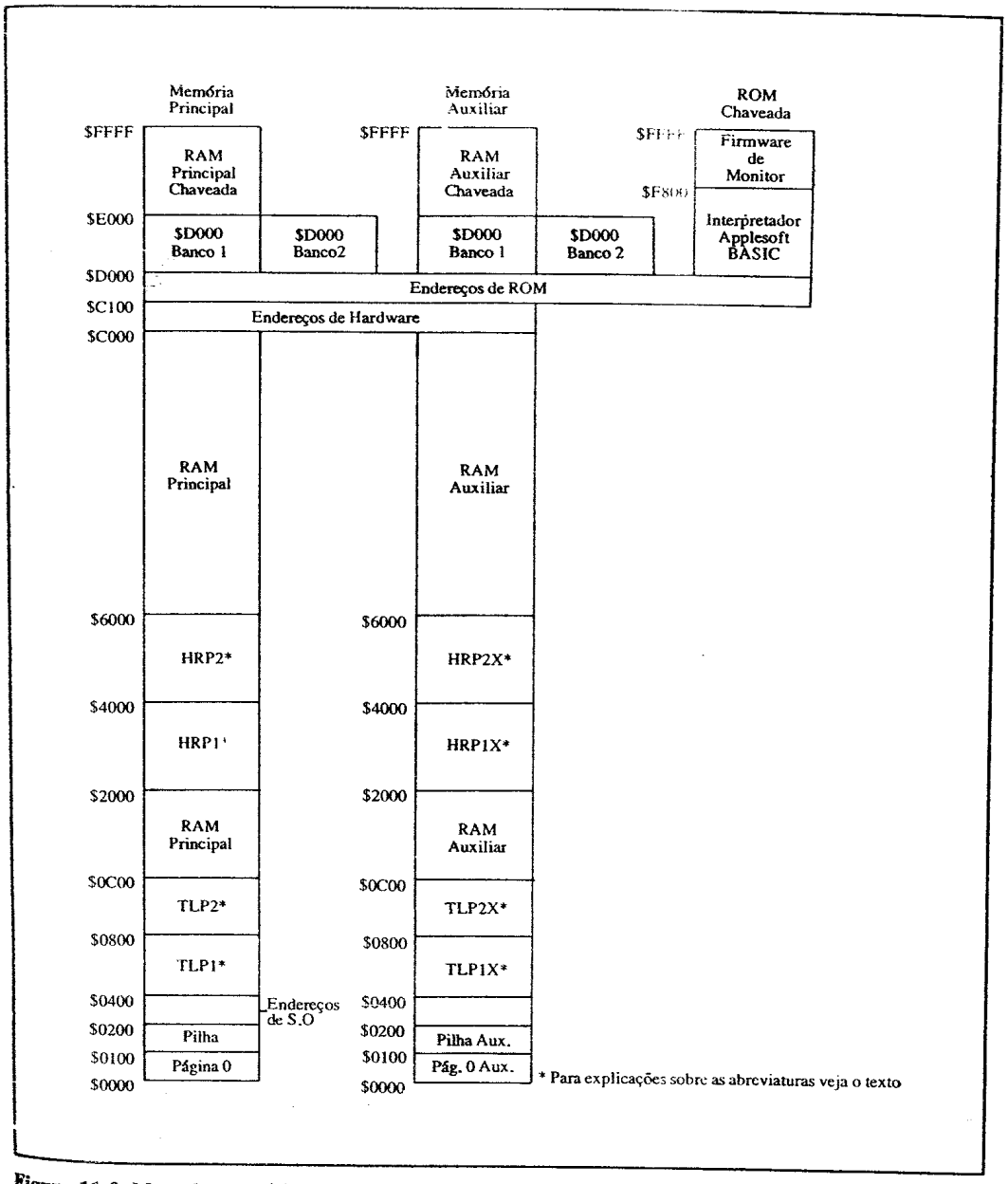


Figura 11.2 Mapa de memória do Apple IIc e do Apple IIe totalmente expandido.

- Certifique-se de que seu programa não requer o uso do Applesoft BASIC. Se o programa for completamente independente do BASIC, você pode utilizar livremente os endereços de $56 a $FF – quase a totalidade da Página Zero.

- Desative as interrupções com uma instrução SEI e, então, salve o conteúdo de parte da Página Zero em outro segmento de memória. Você poderá utilizar essa parte livre da Página Zero. Após a utilização, pode restaurar o conteúdo original com outra movimentação de bloco, e reativar as interrupções, com uma instrução CLI, para permitir que o Sistema Operacional volte ao funcionamento normal.

- Com o auxílio dos *soft switches* RAMRD e RAMWRT (e outros, que serão mencionados mais adiante), você poderá escrever seu programa de modo que ele resida na memória auxiliar.

Tabela 11.2 Posições da Página Zero Disponíveis para Programas do Usuário

$06—$09
$19—$1F
$CE—$CF
$D6—$D7
$E3
$EB—$EF
$F9—$FF

Desta forma você não precisará se preocupar com quaisquer dos registros da Página Zero da memória principal, utilizados pelo Applesoft, ProDOS ou monitor. Como outra alternativa, pode usar o *soft switch* ALTZP para selecionar a Página Zero da memória auxiliar. Porém, se adotar este procedimento, lembre-se das advertências a respeito do ALTZP, fornecidas anteriormente neste capítulo.

ENDEREÇOS DE $0100 A $01FF (PÁGINA 1: A PILHA)

A pilha é localizada na Página $01 da memória de seu computador – isto é, nos endereços de $0100 a $01FF. (Uma explanação detalhada da pilha de hardware foi apresentada no Capítulo 5.)

Neste capítulo, aprendemos que existem, na realidade, duas pilhas no Apple IIc ou no IIe expandido – uma residindo na memória principal e outra na auxiliar. Três "soft switches" – RAMRD, RAMWRT e ALTZP – podem ser usados para chaveamento entre as duas pilhas, durante programas em assembly. Entretanto, deve-se ter muito cuidado ao utilizar esses *switches* para mudar a posição da pilha, pois se efetuarmos essa

movimentação sem as precauções necessárias, poderemos arruinar um programa em assembly.

ENDEREÇOS DE $0200 A $02FF (PÁGINA 2: O BUFFER DE ENTRADA)

Tanto o Applesoft BASIC quanto o monitor usam os endereços de memória de $0200 a $02FF como um buffer para o teclado. Este buffer é usado pelo Applesoft BASIC e pela rotina GETLN, residente no Apple IIc/IIe. Como este buffer ocupa uma página inteira de memória – especificamente a Página 2 – tem uma extensão de 256 bytes.

Nos programas em assembly que não requerem o uso de seqüências de entrada digitadas, este segmento de memória pode ser utilizado para outras finalidades. Se forem necessárias seqüências digitadas, mas que nunca atingem comprimento superior a 256 bytes, a parte superior da Página2 poderá ser utilizada no programa. Na memória auxiliar, o bloco de memória de $0200 a $02FF pode ser usado como RAM livre.

ENDEREÇOS DE $0300 A $03FF (PÁGINA 3: VETORES E ENDEREÇOS DE LIGAÇÃO)

O sistema operacional de disco ProDOS e o monitor usam o espaço de endereçamento de $03F0 a $03FF para determinados endereços de ligação (link addresses) e vetores. Os endereços de $0300 a $03FE são, geralmente, disponíveis para os programas em assembly escritos pelo usuário. Na memória auxiliar, o bloco de memória de $0300 a $03FF pode ser usado como RAM livre.

ENDEREÇOS DE $0400 A $07FF TEXTO E BAIXA RESOLUÇÃO, PÁGINA 1

O segmento de memória que se estende de $0400 até $07FF é a *área de apresentação na tela*, primária, nos Apple IIc e IIe. Quando no modo de texto em 40 colunas, os registros de memória de $0400 a $07FF são usados para acomodar os caracteres que aparecem na tela de seu monitor de vídeo. Quando não usado para gerar apresentações na tela, esse bloco pode ser usado como RAM livre em programas em linguagem assembly.

Sendo o banco de memória $0400-$07FF a principal área de tela no Apple IIc/IIe, é às vezes chamado de *Página 1 de Texto e Baixa Resolução* ou *TLP1* (Text and

Low-Resolution Page 1). A palavra "página", neste caso, refere-se a uma apresentação na tela.

Para criar apresentação de texto em 80 colunas, o Apple IIc/IIe normalmente usa duas áreas de tela, usualmente TLP1 e TLP1X. Alternativamente, as áreas TLP2 e TLP2X podem ser usadas.

Quando no modo de 80 colunas, a apresentação é obtida pelo intercalamento do conteúdo de um buffer de tela da memória principal com o conteúdo do buffer de tela correspondente da memória auxiliar, ou seja, metade dos caracteres presentes na tela vêm do buffer de tela da memória principal e metade do buffer de tela da memória auxiliar, de forma intercalada.

Haverá uma explicação mais detalhada sobre este processo, em capítulos que tratam especificamente de apresentações no vídeo e gráficos Apple.

ENDEREÇOS DE $0800 A $0BFF
TEXTO E BAIXA RESOLUÇÃO, PÁGINA 2

Como vimos, TLP2 é um buffer de tela alternativo, que pode ser usado para gerar apresentação de texto em 40 colunas. Este buffer é, às vezes, utilizado em conjunto com o TLP1, para permitir a um programa "pular" instantaneamente de uma tela a outra. Se o bloco de memória $0800-$0BFF não é necessário em um programa, pode ser usado como RAM livre.

Na memória auxiliar, a área de apresentação na tela que corresponde a TLP2 é chamada TLP2X, que também pode ser usada como RAM livre, quando não necessária para apresentações na tela.

Além dos buffer de texto e baixa resolução acima descritos, quatro buffers de tela de alta resolução são também disponíveis no Apple IIc ou IIe expandido. As posições e descrições destas áreas de memória de tela de alta resolução serão fornecidas a seguir, nesta secção.

São usados *soft switches* para determinar qual área ou áreas de memória serão usadas para gerar apresentações na tela. Esta série de *soft switches* é descrita na Tabela 11.3.

Tabela 11.3 Utilização dos *Soft Switches* para Seleção dos Buffers para Apresentações na Tela

Nome do *switch*	Posição	Como é utilizado
RAMRD	$C002	A gravação de qualquer valor em $C002 desativa o RAMRD e seleciona a memória principal para leitura
RAMRD	$C003	A gravação de qualquer valor em $C003 ativa o RAMRD e seleciona a memória auxiliar para leitura.
RAMWRT	$C004	A gravação de qualquer valor em $C004 desativa o RAMWRT e seleciona a memória principal para gravação.
RAMWRT	$C005	A gravação de qualquer valor em $C005 ativa o RAMWRT e seleciona a memória auxiliar para gravação.
HIRES	$C056	A gravação de qualquer valor em $C056 desativa o HIRES. Quando o HIRES está desativado, uma página de texto e baixa resolução (TLP) é apresentada, e o PAGE2 chaveia entre TLP1 e TLP2.
HIRES	$C057	A gravação de qualquer valor em $C057 ativa o HIRES. Quando o HIRES está ativado, uma página de alta resolução é apresentada, e o PAGE2 chaveia entre HRP1 e HRP2.
80STORE	$C000	A gravação de qualquer valor em $C000 desativa o 80STORE. Quando o 80STORE está desativado, RAMRD e RAMWRT determinarão se o espaço de tela na memória principal ou na auxilar serão usados para leitura e gravação. O PAGE2 irá selecionar as páginas para apresentação, mas não para leitura e escrita.
80STORE	$C001	A gravação de qualquer valor em $C001 ativa o 80STORE. Quando ativado o 80STORE, o PAGE2 chaveará entre TLP1 e TLP1X (se o HIRES estiver desativado) ou entre HRP1 e HRP1X (se o HIRES estiver ativado). Além disso, RAMRD e RAMWRT serão sobrepujados com relação a apresentações na tela, e as páginas selecionadas pelo HIRES e pelo PAGE2 serão apresentadas.
PAGE2	$C054	A gravação de qualquer valor em $C054 desativa o PAGE2. Quando o PAGE2 está desativado e o HIRES está desativado, o bloco TLP1 é selecionado. Quando o PAGE2 está desativado e o HIRES ativado, o bloco HRP1 é selecionado. Se o 80STORE está desativado, RAMRD e RAMWRT determinarão se o espaço de tela na memória principal ou na auxiliar será usado para leitura e gravação, e o PAGE2 selecionará as páginas para apresentação, mas não para leitura

		ou gravação. Se o 80STORE está ativado, RAMRD e RAMWRT são então sobrepujados com relação a apresentações na tela, e as páginas selecionadas pelo HIRES e pelo PAGE2 serão apresentadas.
PAGE2	$C055	A gravação de qualquer valor em $C055 ativa o PAGE2. Quando o PAGE2 está ativado e o HIRES está desativado, o bloco TLP2 é selecionado. Quando o PAGE2 está ativado e o HIRES ativado também, o bloco HRP2 é selecionado. Se o 80STORE está desativado, RAMRD e RAMWRT determinarão se o espaço de tela na memória principal ou na auxiliar será usado para leitura e gravação, e o PAGE2 selecionará as páginas para apresentação, mas não para leitura ou gravação. Se o 80STORE está ativado, RAMRD e RAMWRT são então sobrepujados com relação a apresentações na tela, e as páginas selecionadas pelo HIRES e pelo PAGE2 serão apresentadas.
RDRAMRD	$C013	O bit 7 de $C013 pode ser lido para determinar se a memória principal (0) ou a auxiliar (1) se encontra em uso para leitura.
RDRAMWRT	$C014	O bit 7 de $C014 pode ser lido para determinar se a memória principal (0) ou a auxiliar (1) se encontra em uso para gravação.
RDHIRES	$C01D	O bit 7 de $C01D pode ser lido para determinar se o HIRES está ativado (1) ou desativado (0).
RD80STORE	$C018	O bit 7 de $C018 pode ser lido para determinar se o 80STORE está ativado (1) ou desativado (0).
RDPAGE2	$C01C	O bit 7 de $C01C pode ser lido para determinar se o PAGE2 está ativado (1) ou desativado (0).

ENDEREÇOS DE $0C00 A $1FFF
RAM LIVRE

Os registros de memória de $0C00 a $1FFF são RAM Livre, podendo assim ser usados para qualquer finalidade, em programas do usuário. Como é considerada uma boa prática, em programação, a separação das áreas usadas para programas das áreas usadas para dados, é freqüente a utilização do bloco de memória $0C00 - $1FFF para tabelas de dados, nos programas Apple IIc/IIe. O bloco maior de RAM, compreendido entre $6000 e $BFFF (ou entre $2000 ou $4000 e $BFFF, se não for necessária a utilização de gráficos de alta resolução), pode então ser usado para armazenamento das partes compostas por código executável dos programas em linguagem de máquina.

Na memória auxiliar, da mesma forma que na principal, o banco $0C000–$1FFF é disponível para utilização como RAM livre.

ENDEREÇOS DE $2000 A $5FFF
ALTA RESOLUÇÃO, PÁGINAS 1 e 2

O bloco de memória HRP1, que se estende de $2000 a $3FFF, é o buffer de memória de tela mais freqüentemente usado para criar apresentações em alta resolução. Muitos programas com gráficos de alta resolução utilizam, também, a área de memória de tela HRP2, compreendida entre $4000 e $5FFF. Quando estes blocos não são necessários para gerar apresentações na tela, podem ser usados como RAM livre.

Na memória auxiliar, o bloco que se estende entre $2000 e $3FFF é chamado HRP1X, e o bloco entre $4000 e $5FFF é chamado HRP2X. Estes bancos também podem ser usados como RAM Livre, quando não necessários para geração de gráficos.

O Apple IIc/IIe pode gerar gráficos em cores de alta resolução dupla, intercalando os buffers de tela de alta resolução, da memória principal, com os buffers correspondentes da memória auxiliar. Os gráficos de alta resolução dupla serão estudados em capítulo próprio, neste livro.

ENDEREÇOS DE $6000 A $BFFF
RAM LIVRE

Como já foi mencionado, o banco de memória de $6000 a $BFFF na memória principal é chamado de RAM Principal, e o banco de 48K que se estende de $6000 a $BFFF na memória auxiliar é chamado de RAM Auxiliar. Estes são os principais bancos de RAM nos Apple IIc e IIe, ambos disponíveis para programas do usuário.

ENDEREÇOS DE $C000 A $C0FF
ENDEREÇOS DE HARDWARE

Os registros de memória entre $C000 e $C0FF são utilizados para diversos tipos de funções de hardware:

- A *entrada de dados* (data input) pode ser lida para determinar se uma tecla foi pressionada no teclado, e que tecla é essa.

- Um conjunto de *entradas de indicadores* (flag inputs) pode ser usado para ler os controles manuais, as teclas MAÇÃ ABERTA e MAÇÃ FECHADA do teclado, ou o botão do mouse Apple.

- Um par de *saídas strobe* (strobe outputs) pode determinar se uma tecla foi pressionada, e que tecla é essa, e pode também ler os *game paddle*.

- Um *interruptor "de alavanca"* (toggle switch) opera o alto-falante do Apple IIc/IIe.

- Uma série de *soft switches*, alguns dos quais já foram mencionados neste capítulo. Outros, também residentes no bloco $C000–$C0FF, serão descritos nos Capítulos 12 e 13.

ENDEREÇOS DE $C100 A $CFFF
ENDEREÇOS DE ROM

O bloco de $C100 a $CFFF é exclusivamente dedicado a ROM. Os endereços de ROM residentes nesse bloco incluem:

- Pontos de entrada (entry points) para acesso aos conectores seriais – Serial Port 1 e Serial Port 2 - no IIc.

- Pontos de entrada para acesso à saída de vídeo, saída de vídeo *enhanced* e entrada-e-saída (I/O) geral.

- No Apple IIc, ou no IIe equipado com mouse, pontos de entrada para acesso ao firmware do mouse.

- Pontos de entrada para acessos a disco.

ENDEREÇOS DE RAM, DE $D000 A $FFFF

Quando os endereços de $D000 a $FFFF são usados como RAM, o bloco de memória que se estende de $D000 a $DFFF pode ser usado de quatro formas diferentes: como *Memória Principal $D000 Banco 1, Memória Principal $D000 Banco 2, Memória Auxiliar $D000 Banco 1 e Memória Auxiliar $D000 Banco 2*.

O chaveamento entre estes quatro bancos de memória é controlado por *soft switches*, através das técnicas anteriormente descritas neste capítulo.

O bloco de memória de $E000 a $FFFF pode ser usado de duas maneiras: como *RAM chaveada principal* ou *RAM chaveada auxiliar*. O chaveamento também é controlado por *soft switches*, cuja descrição e técnicas de utilização foram apresentadas na Tabela 11.1.

ENDEREÇOS DE ROM, DE $D000 A $F7FF
O INTERPRETADOR BASIC

O interpretador Applesoft BASIC, residente nos Apple IIc e IIe, encontra-se no bloco de memória de $D000 a $F800. Ele somente pode ser usado quando a ROM é chaveada para este bloco de memória, o que pode ser efetuado através dos *soft switches* presentes na Tabela 11.1.

ENDEREÇOS DE ROM, DE $F800 A $FFFF
O MONITOR

Quando a ROM é chaveada para o bloco de memória $F800-$FFFF, este é o lugar onde reside o monitor. Como o monitor é muito conveniente para a escrita e depuração de programas em assembly, é recomendável evitar a colocação de código escrito pelo usuário neste segmento de memória.

PROGRAMAÇÃO EM LINGUAGEM ASSEMBLY SOB ProDOS

Quando da instrução do Apple IIc, um novo sistema operacional, o ProDOS, veio substituir o antigo sistema operacional de disco da antiga série II da Apple, o DOS 3.3. A maioria do software profissional para o Apple IIc e para o Apple IIe é agora escrita sob ProDOS, e é de bom senso escrevermos nossos programas sob ProDOS, também.

O ProDOS não é simplesmente um sistema operacional de disco. É um sistema operacional completo, que permite ao programador em assembly o gerenciamento da maioria dos recursos apresentados pelos Apple IIc e IIe. O ProDOS trabalha principalmente como um sistema operacional de disco, mas também manipula interrupções. Além disso, possui uma ferramenta de gerenciamento de memória, chamada *Interface de Linguagem de Máquina, ou* (Machine Language Interface), que é a parte do

sistema operacional que recebe, valida e emite comandos de operação do sistema.

Em linguagem assembly, os comandos MLI são usados de forma bem similar à forma pela qual os comandos DOS são usados em BASIC. Nos programas em assembly, chamadas MLI podem ser usadas para abrir e fechar arquivos, criá-los, eliminá-los e realizar várias outras funções relacionadas ao disco. (Na realidade, quase todos os comandos ProDOS disponíveis em BASIC – e alguns que não o são – estão disponíveis através da MLI.)

A discussão completa do ProDOS e da interface de Linguagem de Máquina encontra-se além do escopo deste capítulo. Se desejar aprender mais a respeito do ProDOS, leia *ProDOS User's Manual*, o *BASIC Programming With ProDOS* e, é claro, o *ProDOS Technical Reference Manual*.

O propósito desta secção é o de indicar onde o ProDOS reside, após ser carregado na memória do computador, de forma a evitar que seja sobrescrito por programas do usuário.

UM MAPA DE MEMÓRIA DO ProDOS

Quando o ProDOS é carregado na memória de um Apple IIc ou Apple IIe, ele ocupa diferentes extensões de RAM, dependendo da configuração do computador. Num Apple IIc ou num IIe equipado com um cartão de 80 colunas expandido, parte do espaço consumido pelo ProDOS situa-se na memória principal e parte na auxiliar.

O ProDOS é usualmente carregado na memória junto com um arquivo chamado BASIC.SYSTEM, que permite ao ProDOS processar comandos BASIC. Ao serem estes carregados simultaneamente, é consumido um espaço adicional de memória, pois o arquivo BASIC.SYSTEM, em um disco ProDOS auto-inicializável (startup disk), inclui um interpretador de comandos.

A Figura 11.3 é um mapa de memória que mostra os blocos de memória consumidos pelo ProDOS e pelos arquivos BASIC.SYSTEM, quando o ProDOS é carregado na memória a partir de um *startup disk*. Os programas em linguagem assembly, projetados para execução sob controle do ProDOS, não devem invadir as áreas sombreadas na Figura 11.3

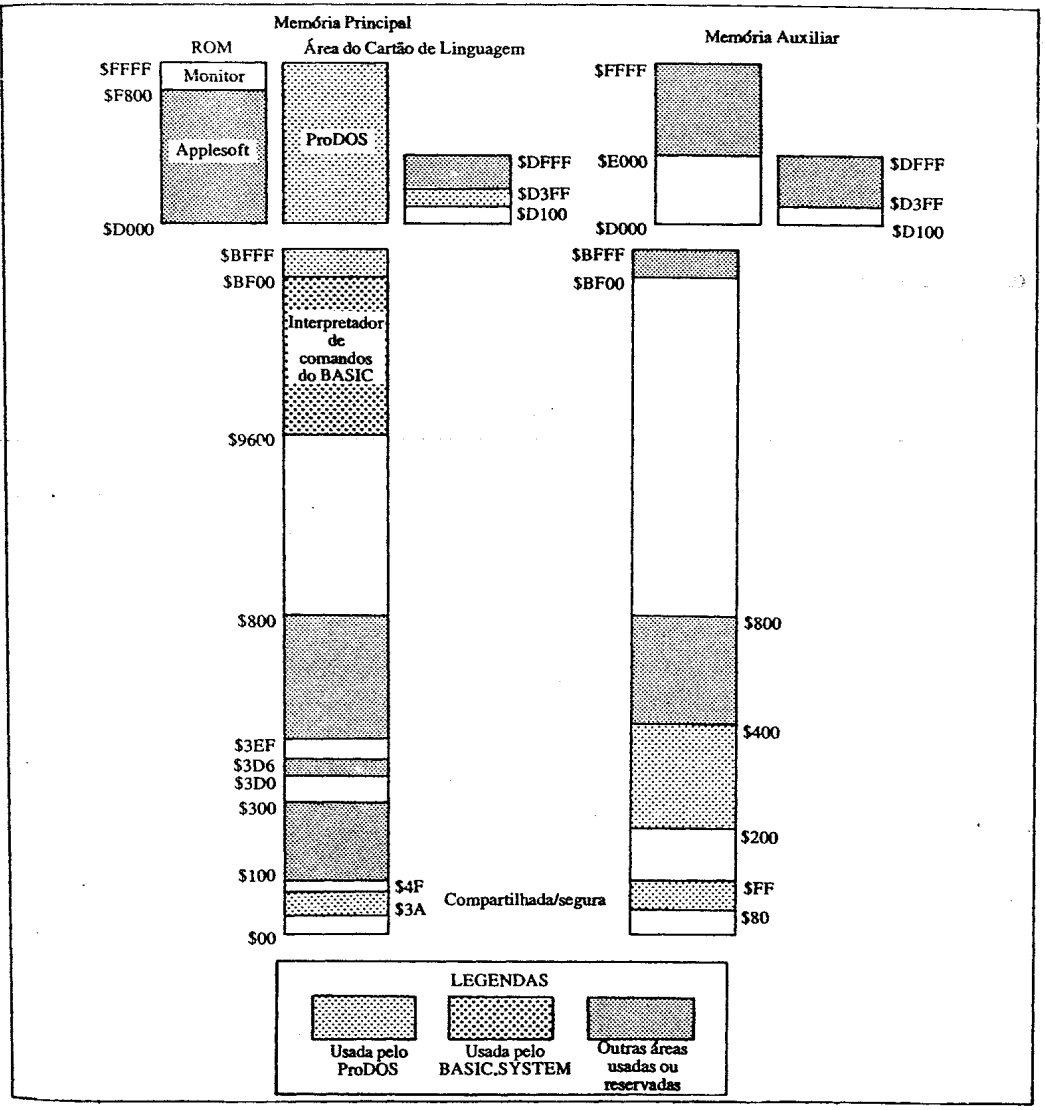


Figura 11.3 Mapa de Memória –ProDOS e BASIC.

REQUISITOS DE MEMÓRIA DOS ASSEMBLERS

Ao escrever um programa em assembly, você deve ter em mente a área onde seu sistema assembler-editor reside na memória do computador, de forma a evitar que o programa danifique o assembler durante a edição ou compilação.

MEMÓRIA PARA O MERLIN

O assembler Merlin Pro é uma peça de software complexa, consumindo uma boa porção de RAM, tanto da memória principal quanto da auxiliar. Porém, o Merlin provê uma função de encadeamento (link) que permite conectar longos programas, compilando-os diretamente no disco. Detalhes sobre a utilização desta função podem ser encontrados no manual de instruções do Merlin Pro.

MAPEANDO O APPLE ProDOS ASSEMBLER

A Figura 11.4 representa o mapa de memória do assembler-editor Apple ProDOS. O sistema da Apple consome menos memória que o Merlin, e todo o espaço ocupado situa-se na memória principal. Desta forma, o Apple ProDOS Assembler mantém livres os 64K da memória auxiliar para programas do usuário.

Uma característica notável do Apple ProDOS é que o editor e o assembler não residem na memória simultaneamente. Portanto, ao escrever um programa em código-fonte usando o sistema, ele deverá, primeiramente, ser gravado em disco, antes de ser compilado. Este processo consome tempo, mas economiza memória.

REQUISITOS DE MEMÓRIA DO ORCA/M ASSEMBLER

O ORCA/M realiza um intercâmbio com o disco de forma ainda mais intensa que o realizado pelo sistema assembler-editor Apple. Devido a isto – e à forma elegante com que o ORCA/M é projetado – é possível ao usuário escrever longos programas em assembly, colocando-os quase que em qualquer local da memória. Segundo a The Byte Works, que produz o ORCA/M, a lista a seguir representa todas as restrições de memória do sistema:

- O sistema ORCA/M roda sob o controle de um programa intitulado ORCA.HOST. Este programa, que ocupa os endereços de $0800 a $1FFF, precisa permanecer na memória durante todo o tempo de utilização do ORCA/M. Portanto, não deverá jamais ser sobrescrito por programas do usuário.

- Na Página Zero, as posições de memória de $00 a $7F são disponíveis para o usuário, mas os endereços de $80 a $FF devem ser evitados, pois são utilizados pelo sistema ORCA/M.

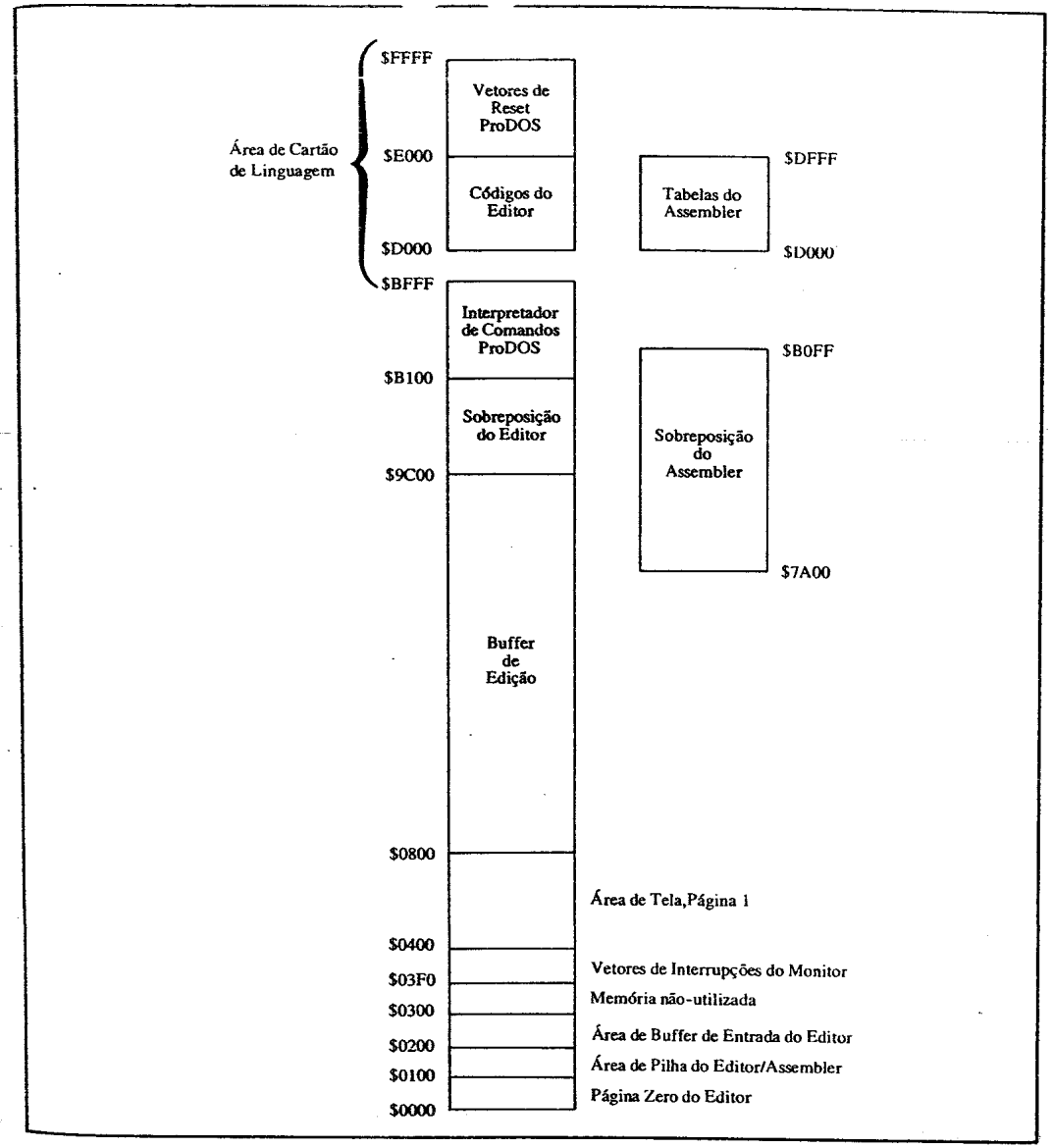


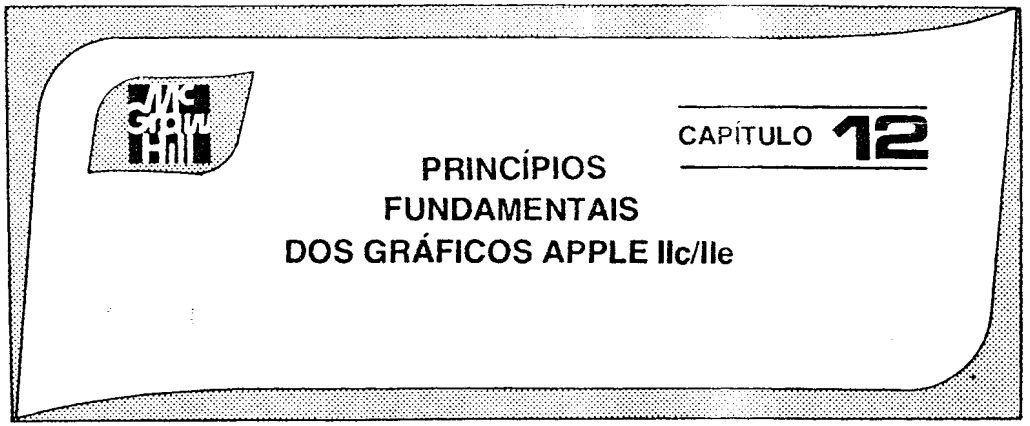
Figura 11.4 Mapa de Memória do Apple ProDOS Assembler.

- O buffer de linha do ProDOS usa os registros de memória de $200 a $2FF, e o ORCA/M usa o buffer de linha do ProDOS, quando em modo de comando. Até que um programa tenha sido depurado e esteja pronto para ser rodado, é recomendável evitar o uso daqueles endereços.

- Cartões com relógio (clock) não-standard algumas vezes utilizam-se dos registros de memória de $300 a $3FF; portanto, programas em assembly que fazem uso desses registros podem interferir na operação de tais cartões. (Esta restrição aplica-se a programas escritos com *qualquer* assembler.)

CONCLUSÃO

Uma vez familiarizado com os requisitos (e as limitações) de memória de seu computador, sistema operacional e sistema assembler-editor, você tem todas as informações necessárias para usar a memória de forma segura e eficiente em programas em linguagem assembly.



CAPÍTULO 12

PRINCÍPIOS FUNDAMENTAIS DOS GRÁFICOS APPLE IIc/IIe

Se você não dedica a maior parte de seu tempo livre ao contato com outros programadores, provavelmente deve ter percebido que a maioria das pessoas não se impressiona muito com algoritmos elegantes ou longas listagens de código em linguagem assembly.

No entanto, se você souber criar programas gráficos – especialmente os de ação rápida, bem animados, que somente podem ser criados em linguagem assembly – poderá realmente entusiasmar seus amigos. Se esta proposta lhe fascina, vá em frente, pois o restante deste livro é dedicado a programas gráficos em linguagem assembly.

Neste capítulo, aprenderemos como o Apple IIc e o Apple IIe geram texto e gráficos. No Capítulo 13, você poderá digitar e rodar programas que demonstram o modo gráfico de baixa resolução, e verá também como os controladores de jogos e o mouse podem ser usados nos programas em assembly. O Capítulo 14 irá lhe mostrar como personalizar um jogo de caracteres, como apresentar caracteres de tamanho normal e de tamanho *manchete* em telas de alta resolução, e como escrever programas usando gráficos de dupla alta resolução.

MODOS GRÁFICO E DE TEXTO

Se você tem um Apple IIc, ou um modelo recente de Apple IIe, com um cartão de 80 colunas com expansão de memória, seu computador pode ser usado em seis modos básicos de texto e gráficos:

- Texto em quarenta colunas.
- Texto em oitenta colunas.

- Gráficos de baixa resolução.
- Gráficos de dupla baixa resolução.
- Gráficos de alta resolução.
- Gráficos de dupla alta resolução.

Estes seis modos serão discutidos, pelo menos brevemente, neste capítulo. Entretanto, focalizaremos nossa atenção principalmente nos dois modos mais comumente usados na programação gráfica do Apple IIc/IIe: os modos gráficos normais de baixa e alta resolução.

MODOS DE TEXTO EM 40 E 80 COLUNAS

O Apple IIe *standard*, sem um cartão de texto em 80 colunas ou um cartão Apple Color Adaptor, apresenta somente um modo de texto: o modo 40 colunas, que gera apresentação de texto em 24 linhas de 40 colunas. O Apple IIc, ou o IIe equipado com o cartão de 80 colunas, pode também gerar apresentação de texto em 24 linhas de 80 colunas.

Como vimos no Capítulo 11, os *soft switches* podem ser usados para determinar se a apresentação do texto, em um programa, dar-se-á em 40 ou 80 colunas. Os *soft switches* podem, também, ser usados para promover o chaveamento entre textos e gráficos, e para estabelecer uma "janela de texto" de 4 linhas por 40 ou 80 colunas, na parte inferior das telas gráficas de alta ou baixa resolução.

MAPEAMENTO DA MEMÓRIA

Seu Apple IIc/IIe usa uma técnica conhecida por *mapeamento da memória* para criação das apresentações gráficas ou de texto. Isto significa que, através do armazenamento de valores específicos em determinados blocos da memória do computador, podemos controlar a tela. Quando seu computador se encontra em um dos modos de texto, por exemplo, você pode armazenar caracteres ASCII (American Standard Code for Information Interchange) – modificados para computadores Apple – em um bloco específico de RAM. Feito isto, o sistema operacional do computador converterá cada código numérico em uma letra, número ou caractere especial, e apresentará cada um deles em uma posição de tela determinada pelo byte específico da RAM na qual o código numérico do caractere havia sido armazenado.

Normalmente, os códigos Apple ASCII que geram texto em 40 colunas são armazenados na área de RAM chamada Texto e Baixa Resolução, Página 1, que se estende do endereço $400 ao $7FF. Alternativamente, aqueles códigos podem ser armazenados no bloco Texto e Baixa Resolução, Página 2, entre $800 e $BFF.

Se o seu computador é um Apple IIc, ou um IIe equipado com um cartão de 80 colunas expandido, existem duas outras áreas de RAM que podem ser usadas como memória de tela para texto em 40 colunas: Texto e Baixa Resolução, Páginas 1X e 2X, respectivamente de $400 a $7FF e de $800 a $BFF, na memória auxiliar.

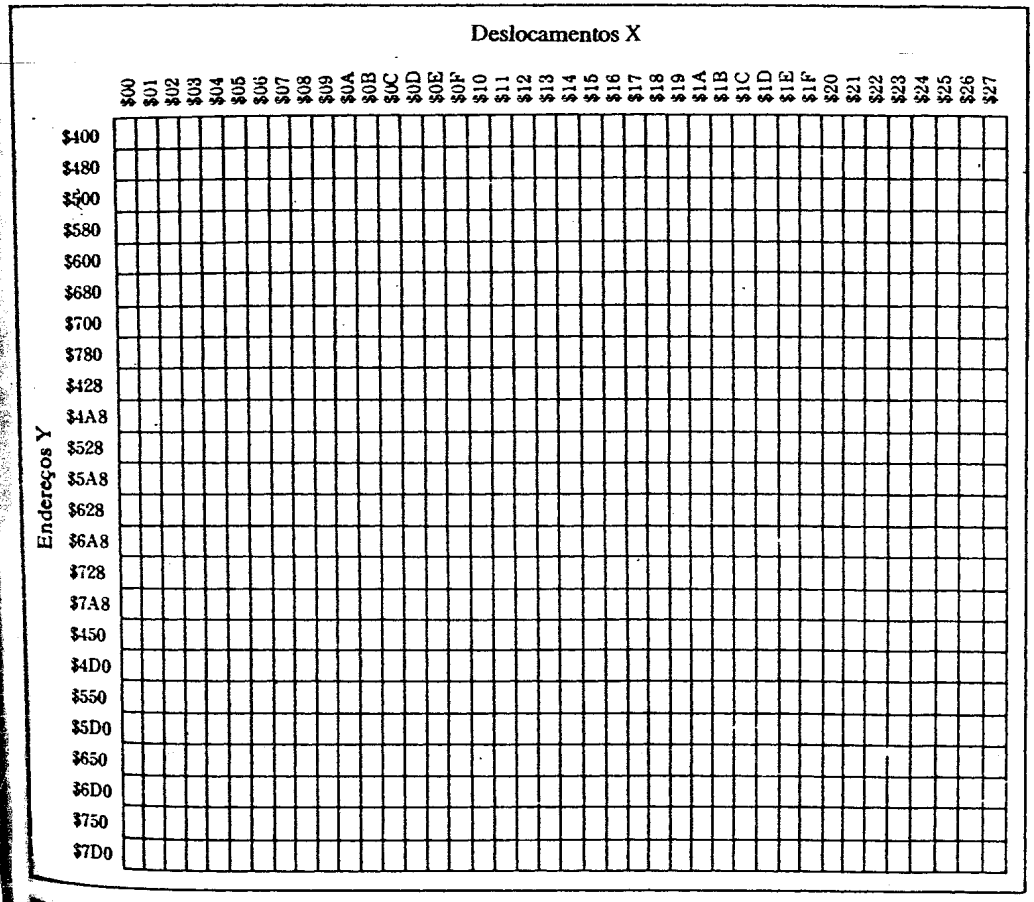


Figura 12.1 Mapa da Tela de Texto em 40 Colunas.

Tanto no IIc como no IIe, os *soft switches* podem ser usados para designar as áreas da memória principal ou auxiliar a serem usadas para o armazenamento de texto ou

informação gráfica. As funções e os endereços destes *switches*, em sua maioria, foram listadas no Capítulo 11, onde também foram apresentados os mapas de memória que indicavam os segmentos de RAM que podiam ser usados para armazenamento de texto e informação gráfica.

A Figura 12.1 é um mapa de memória da Página de Vídeo 1, a área de memória mais freqüentemente usada para apresentação de texto em 40 colunas.

CRIANDO TEXTO EM 80 COLUNAS

Quando o Apple IIc (ou IIe equipado com cartão de texto em 80 colunas ou cartão de

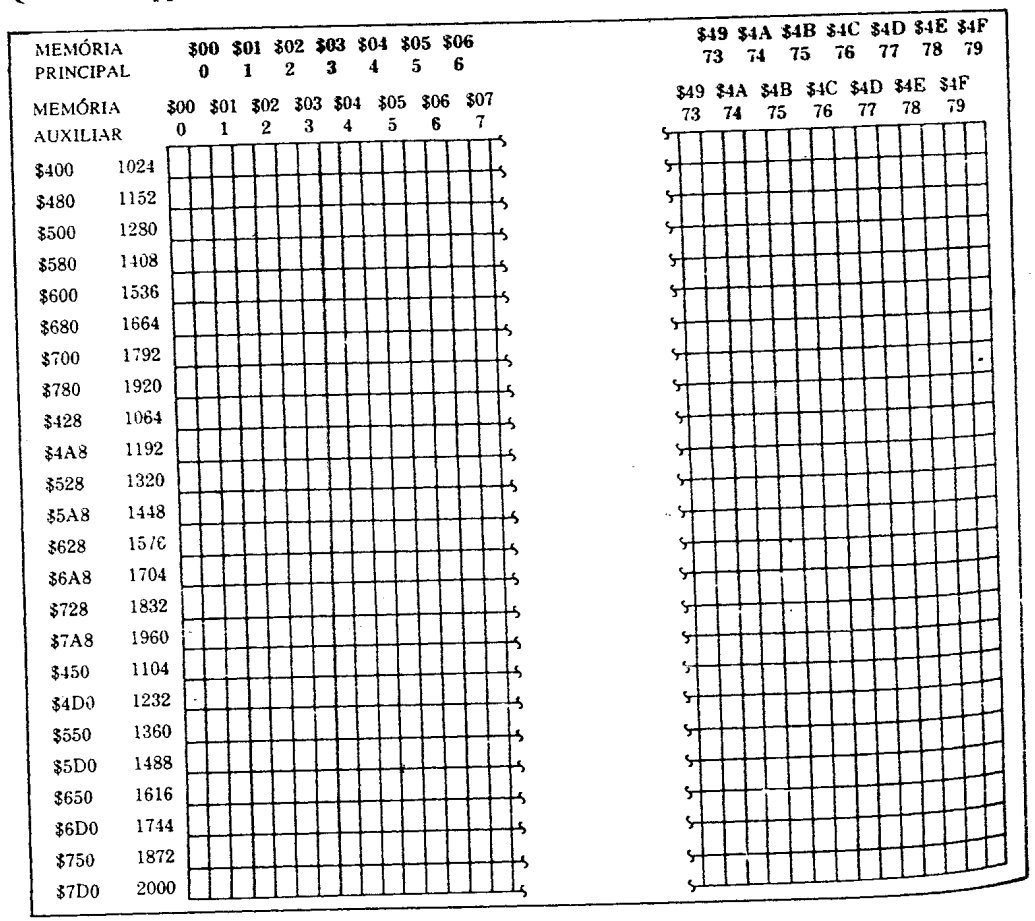


Figura 12.2 Mapa da Tela de Texto em 80 colunas.

cores) se encontra no modo 80 colunas, os códigos ASCII para os caracteres localizados nas colunas de numeração par (sendo a primeira coluna numerada como coluna 0), são armazenados na Página de Vídeo 1, na memória principal.

Os códigos para os caracteres localizados nas colunas ímpares (com a segunda coluna numerada como Coluna 0) são armazenados na Página de Vídeo 1X, na memória auxiliar. Ao serem apresentados em uma tela de 80 colunas, os caracteres desses dois blocos de memória são, automaticamente, intercalados, como ilustra a Figura 12.2.

Felizmente, não é necessário preocuparmo-nos com este processo de intercalação, ao escrevermos programas; ele é normalmente realizado pelo sistema operacional do computador.

GRÁFICOS DE BAIXA RESOLUÇÃO

O sistema de mapeamento da memória usado para gráficos de baixa resolução é bastante similar ao usado para texto em 40 colunas. Uma tela gráfica de baixa resolução, no Apple IIc/IIe, é formada por uma grade de retângulos coloridos chamados *elementos de figura, elementos de retícula* ou *pixels*. Esta matriz de tijolos coloridos possui 40 elementos de largura por 48 de altura – em outras palavras, 40 colunas por 48 linhas.

Para colocar o computador em modo gráfico de baixa resolução, simplesmente ativamos o *soft switch* localizado no endereço de memória $C056, como explicado no Capítulo 11.

Estando o computador em modo gráfico de baixa resolução, podemos criar uma apresentação através do armazenamento de dados nas Páginas de Vídeo 1, 2, 1X e 2X (as mesmas áreas usadas para armazenar códigos Apple ASCII para texto em 40 colunas). Quando no modo gráfico, entretanto, o computador não interpretará os dados armazenados naquelas áreas como códigos de caracteres, e sim como cores, que serão apresentadas na tela.

Em baixa resolução, o Apple IIc/IIe pode apresentar 16 cores incluindo o branco e o preto. Estas cores, e os códigos usados para gerá-las, encontram-se na Tabela 12.1.

A Figura 12.3 mostra como a Página 1 é disposta quando usada para apresentar uma tela gráfica de baixa resolução. Cada byte armazenado na memória da tela é usado para gerar dois *pixels* coloridos, um acima do outro.

Assim, há o dobro de elementos em uma tela gráfica de baixa resolução, com

relação a uma tela de texto em 40 colunas: um gráfico de baixa resolução apresenta 40 colunas por 48 linhas, ao passo que uma tela de texto apresenta 40 colunas por 24 linhas.

GRÁFICOS DE ALTA RESOLUÇÃO

Em modo gráfico de alta resolução (high-res), seu computador apresenta uma grade de 280 pontos de largura por 192 de altura, se utilizada de forma monocromática; a resolução efetiva é de 140 na largura por 192 na altura, quando usamos cores. Em gráficos coloridos de alta resolução, a cor de cada *pixel* na tela pode ser controlada individualmente. Sabendo como usar gráficos de alta resolução, podemos criar figuras e caracteres bastante detalhados e atraentes.

Tabela 12.1 Códigos das Cores em Gráficos de Baixa Resolução

HEXA	COR	HEXA	COR
$0	Preto	$8	Marrom
$1	Magenta	$9	Laranja
$2	Azul-escuro	$A	Cinza 2
$3	Roxo	$B	Rosa
$4	Verde-escuro	$C	Verde-claro
$5	Cinza 1	$D	Amarelo
$6	Azul médio	$E	Turquesa
$7	Azul-claro	$F	Branco

Existem algumas limitações, entretanto, com relação à maneira pela qual as cores podem ser utilizadas em gráficos de alta resolução no Apple IIc/IIe. Somente seis cores reais são disponíveis: preto, branco, violeta, verde, laranja e azul. Usando-as em linhas alternadas, é possível misturá-las, para criar outras cores; porém, a não ser que você seja um expert neste tipo de mescla de cores, ela raramente vale o esforço necessário para executá-la.

Existem também limitações quanto às formas de combinação das cores. Elas serão explicadas no Capítulo 14, dedicado unicamente aos gráficos de alta resolução.

Quando seu computador se encontra em modo gráfico de alta resolução, a área de RAM utilizada mais freqüentemente como memória de tela estende-se do endereço $2000 ao $3FFF – um total de 8192 bytes de memória. Este bloco é algumas vezes chamado de Página Gráfica de Alta Resolução 1. Outro bloco de 8192 bytes, de $4000 a

$5FFF, é também usado com freqüência e chamado de Página Gráfica de Alta Resolução 2.

Se você possui um Apple IIc, ou um IIe equipado com um cartão de 80 colunas expandido, são disponíveis mais dois blocos para memória de tela de alta resolução. Estes dois segmentos de RAM são a Página Gráfica de Alta Resolução, 1X, que abrange os endereços de $2000 a $3FFF na memória auxiliar, e a Página Gráfica de Alta Resolução, 2X, que se estende entre $4000 e $5FFF na memória auxiliar.

Seu Apple utiliza o que é comumente denominado de tela *mapeada em bits* (bit-mapped screen), quando em modo de alta resolução. Numa apresentação na tela mapeada em bits – o tipo de apresentação mais comum em programas comerciais – cada ponto da tela coresponde a um bit na memória do computador.

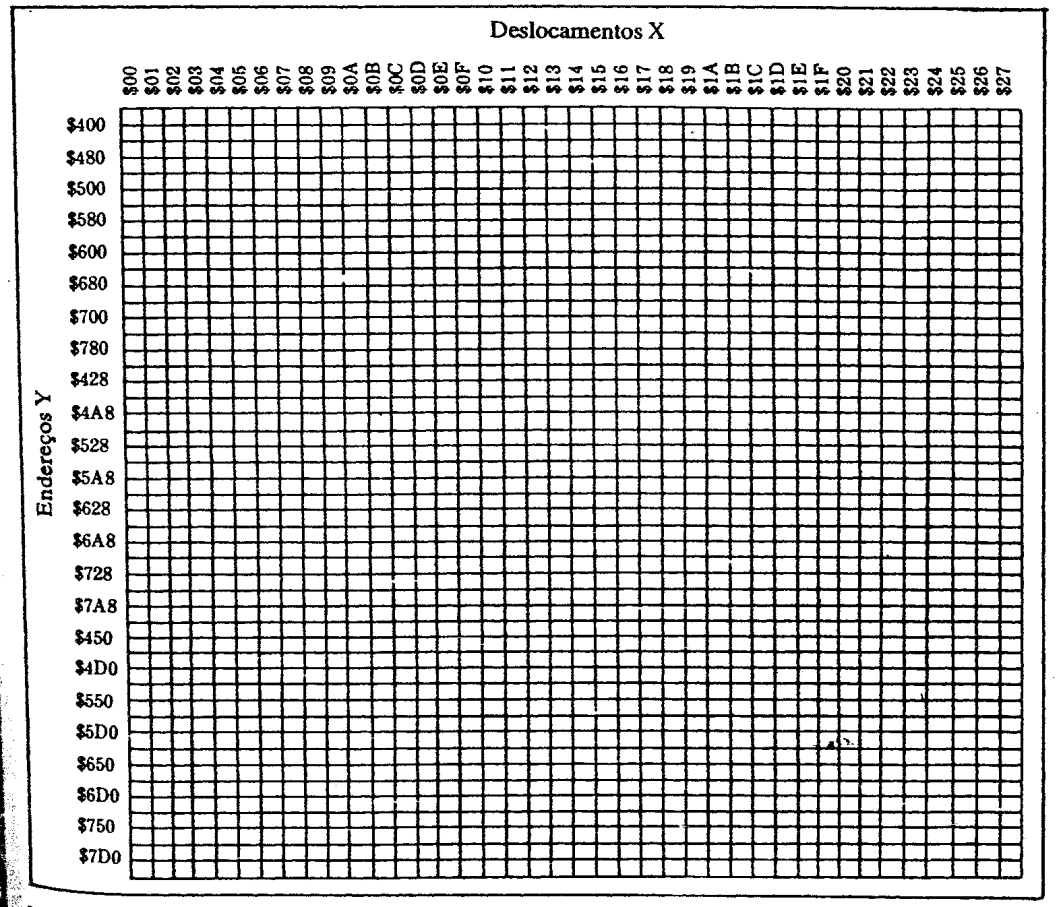


Figura 12.3 Mapa da Tela para Gráficos de Baixa Resolução

Quando um bit está ativado, corresponde geralmente a um ponto aceso na tela; se desativado, o ponto está apagado. Desta forma, cada ponto na tela pode ser controlado individualmente por um programa gráfico com mapeamento em bits.

Nos gráficos de alta resolução do Apple IIc/IIe, cada byte na memória de tela controla sete pontos na tela. Como existem oito bits em um byte, um bit é mantido livre, em cada byte, para ser usado como um bit de status. Ativando ou desativando esse bit, o programador pode controlar as cores geradas pelos outros bits do mesmo byte, quando são apresentados na tela.

Maiores detalhes sobre este processo serão fornecidos no Capítulo 14. Por ora, observemos superficialmente os gráficos de dupla baixa e dupla alta resolução, dois modos não disponíveis para os programadores antes do advento dos Apple IIc e IIe.

GRÁFICOS DE DUPLA BAIXA RESOLUÇÃO

Os gráficos de dupla baixa resolução, como o próprio nome indica, constituem um modo de apresentação com o dobro da resolução que se obtém no modo gráfico comum de baixa resolução. A dupla baixa resolução permite criar uma tela em 16 cores, com 80 *pixels* de largura por 48 de altura.

Se você tem um Apple IIc, ou um Apple IIe com cartão de 80 colunas e cuja placa de circuito impresso seja do tipo Revisão B (ou posterior), poderá escrever e rodar programas gráficos de dupla baixa resolução.

Para identificar o tipo da placa de circuito de seu Apple IIe, abra-o e verifique a marcação impressa logo atrás dos conectores para expansões (slots). Num computador "Revisão B", a letra B estará impressa após o número da peça (por exemplo: 820-0064-B).

Para usar gráficos de dupla baixa (ou alta) resolução em um Apple IIe, é necessário também um cartão de texto em 80 colunas ou um cartão de texto/cores, adequadamente configurados.

Se você tem um cartão de 80 colunas comum (non-color), é necessário conectar os pinos 50 e 55 dele. Se seu cartão é do modelo sem expansão (sem os 64K extras de memória), é necessário usar um ferro de solda para efetuar a conexão dos pinos. Caso você não tenha prática nessas alterações de hardware, será conveniente solicitar o auxílio de um técnico qualificado.

Num cartão non-color expandido, entretanto, a tarefa de interligar os pinos 50 e 55 é bem mais simples: basta conectar o cabo *jumper* que acompanha o cartão. E não é necessário nenhuma alteração de hardware se você tem um Apple IIc ou um IIe com o cartão Text/AppleColor.

Assegurada a capacidade de manuseio de programação em dupla baixa resolução pelo computador, você pode colocá-lo nesse modo através do armazenamento de qualquer valor em todos os *soft switches* situados nos endereços de memória $C050, $C056, $C052, $C00D, $C05E e $C05F. Estes quatro primeiros *switches* selecionam gráficos em lugar de texto, baixa resolução em lugar de alta e gráficos em toda a tela (sem a janela de texto). Estes *switches* também acionam o firmware de 80 colunas do computador. A seguir, é preciso ativar ainda outro *soft switch*, chamado AN3 e situado nas posições de memória $C05E e $C05F. Para ativar gráficos de dupla baixa ou alta resolução, é necessário *desativar* AN3, lendo ou gravando no endereço $C05E. Para desativar os gráficos de dupla baixa ou alta resolução, é preciso *ativar* AN3, lendo ou gravando em $C05F.

Para colocar um Apple adequadamente equipado em modo de dupla baixa resolução, pode ser usada a seguinte seqüência de instruções:

```
STA $C050 ; DESATIVAR MODO DE TEXTO
STA $C056 ; DESATIVAR MODO ALTA RESOLUCAO (HI-RES)
STA $C052 ; DESATIVAR MODO 'MIXED' (OPCIONAL)
STA $C00D ; ATIVAR 80 COLUNAS
STA $C05E ; ATIVAR GRAFICOS BAIXA RESOLUCAO DUPLA
```

Quando no modo de baixa resolução dupla, o Apple IIc/IIe apresenta cores de forma muito similar à utilizada para apresentar texto em 80 colunas. Quando o modo gráfico de dupla baixa resolução é ativado, os dados armazenados em Gráficos de Baixa Resolução, Página 1 (o bloco de memória de $400 a $7FF na memória principal), serão apresentados como blocos de cor nas colunas de numeração par (começando na Coluna 0) da tela.

Estas cores serão intercaladas com as geradas pelos dados armazenados em Gráficos de Baixa Resolução, Página 1X (de $400 a $7FF na memória auxiliar). As cores geradas pela Página 1X serão apresentadas nas colunas ímpares (começando na Coluna 1) da tela.

CÓDIGOS DE CORES DA MEMÓRIA AUXILIAR

Se você decidir usar gráficos de dupla baixa resolução, poderá encontrar um pequeno problema: devido a diferenças de tempo (sincronismo) entre a memória principal e a

auxiliar, os códigos de cores usados para dados de baixa resolução armazenados na memória auxiliar são diferentes dos utilizados para dados armazenados na memória principal.

A Tabela 12.2 apresenta os códigos de cores para dados de gráficos de dupla baixa resolução armazenados na memória auxiliar.

GRÁFICOS DE DUPLA ALTA RESOLUÇÃO

Este modo oferece o dobro da resolução ou o dobro das cores, em relação ao modo gráfico comum de alta resolução. Utilizando gráficos de dupla alta resolução, você pode criar uma tela monocromática de ultra-alta resolução, de 560 pontos de largura por 191 de altura. Alternativamente, você pode criar uma tela em 16 cores, com 140 pontos de largura por 192 de altura, onde é possível apresentar cada ponto em qualquer das 16 cores disponíveis.

Tabela 12.2 Códigos de Cores para Dados de Gráficos de Dupla Baixa Resolução Armazenados na Memória Auxiliar

HEXA	COR	HEXA	COR
$0	Preto	$4	Marrom
$8	Magenta	$C	Laranja
$1	Azul-escuro	$5	Cinza 2
$9	Roxo	$D	Rosa
$2	Verde-escuro	$6	Verde-claro
$A	Cinza 1	$E	Amarelo
$3	Azul médio	$7	Turquesa
$B	Azul-claro	$F	Branco

O mapa de tela usado para gráficos de dupla alta resolução é também similar ao usado para texto em 80 colunas. Quando em modo de alta resolução, o Apple IIc/IIe intercala dados de tela do Display de Alta Resolução, Páginas 1 e 1X, assim apresentando dois bytes no espaço normalmente ocupado por um único. As páginas HRP2 e HRP2X, na memória auxiliar, podem também ser usadas para armazenamento de dados de tela em dupla alta resolução.

Para usar gráficos de dupla alta resolução, é necessário que o Apple IIc ou IIe esteja equipado para gerar gráficos de dupla baixa resolução. Para obter o máximo dos

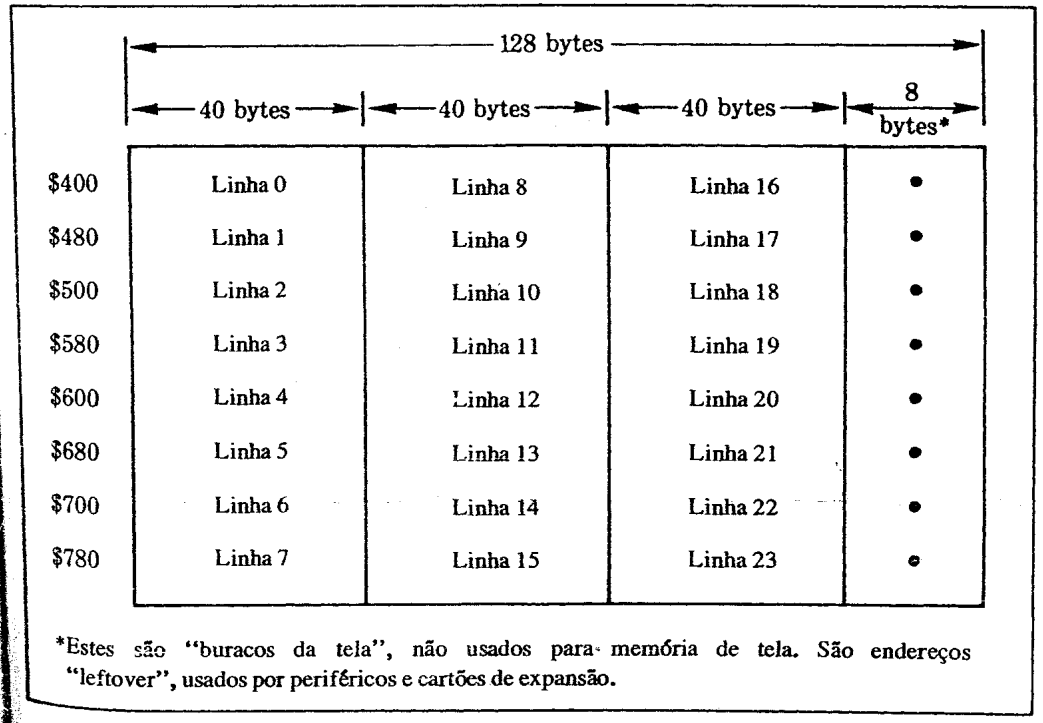
gráficos de dupla alta resolução, é necessário também um monitor monocromático de 80 colunas (para as apresentações de ultra-alta resolução monocromáticas) ou um monitor de 80 colunas RGB de alta qualidade (para as apresentações de dupla alta resolução em cores).

É possível apresentar gráficos em cores em dupla alta resolução em monitores coloridos comuns de 40 colunas ou aparelhos de TV, mas os pontos perdem definição e brilho.

MAPEANDO AS TELAS GRÁFICAS

Agora que já vimos rapidamente os modos gráficos do Apple IIc/IIe, podemos examinar mais detalhadamente os mapas de tela usados para cada um desses modos.

Antes, porém, observemos com mais atenção a apresentação de texto em 40 colunas ilustrada na Figura 12.1. Note que foram usadas seqüências de números hexadecimais para identificar cada linha e cada coluna, no mapa.



*Estes são "buracos da tela", não usados para memória de tela. São endereços "leftover", usados por periféricos e cartões de expansão.

Figura 12.4 Como a memória de tela é armazenada em RAM.

Esses números parecem ser coordenadas de linha e coluna, mas na realidade não o são. Os números à esquerda do mapa são endereços de memória – especificamente, os endereços dos bytes iniciais de cada coluna. Pelo fato de esses endereços representarem posições verticais (ou Y) no mapa de memória, referimo-nos a eles, algumas vezes, como *endereços Y*.

Os números ao longo do topo do mapa representam *deslocamentos* que, adicionados aos endereços Y, permitem determinar o endereço exato de memória de qualquer byte apresentado no mapa. Como esses números são usados para representar posições horizontais (ou X) no mapa, são às vezes chamados de *deslocamentos X* (X offsets).

Com os deslocamentos X e os endereços Y do mapa de tela Apple, a posição exata de qualquer byte do mapa pode ser facilmente determinada: é só somar o deslocamento X do caractere com o seu respectivo endereço Y – isto fornecerá o endereço do caractere na RAM.

Se observarmos os deslocamentos X grafados no topo da Figura 12.1, notaremos que as colunas do mapa são numeradas consecutivamente de $00 a $27 – ou 0 a 39 em notação decimal. Esta numeração torna fácil determinar a coluna de localização do caractere. Não importa em que linha o caractere esteja; se ele está em uma dada coluna na tela, o deslocamento X será sempre o mesmo.

Os endereços Y na Figura 12.1 não são numerados consecutivamente, e sim dispostos em três grupos de oito linhas. As primeiras oito linhas no mapa são numeradas de $400 a $780; as próximas oito linhas, de $428 a $7A8; e as oito linhas inferiores, de $450 a $7D0.

À primeira vista, este sistema de numeração pode parecer estranho. Mas, na realidade, faz algum sentido: se a memória de tela do computador não fosse dividida, "fatiada", apresentando-se como um bloco sólido de RAM de 960 bytes, os deslocamentos X e os endereços Y não poderiam ser utilizados para localização de caracteres na tela.

Caracteres na mesma coluna dificilmente teriam o mesmo deslocamento X, e caracteres na mesma linha não necessariamente apresentariam o mesmo endereço Y. Assim, o sistema um pouco confuso de deslocamentos X e endereços Y, utilizado para a tela de 40 colunas, é preferível a... nenhum sistema!

A Figura 12.4 mostra como se apresenta o mapa de tela de 40 colunas quando disposto como um bloco único de memória. Como podemos notar, o sistema de

"endereços e deslocamentos" projetado para seu computador é bastante eficiente em termos de memória. Ele permite colocar um mapa de tela de 960 bytes em 1024 bytes de memória, deixando sem uso apenas 64 bytes. E cada caractere na tela pode ser localizado pela adição de um deslocamento de coluna com um endereço de linha.

Existe mais um fato a respeito do mapeamento de tela que vale a pena mencionar. Os sistemas usados para mapeamento de gráficos de baixa e alta resolução são baseados nos mesmos princípios usados para mapeamento da tela de texto em 40 colunas:

A Figura 12.3 mostra que o mapa de tela usado para gráficos de baixa resolução é o mesmo usado para texto em 40 colunas; a única diferença é que cada byte na tela foi dividido horizontalmente em dois *pixels*.

CAPÍTULO 13

CONTROLADORES DE JOGOS E O APPLE MOUSE

Para tornar-se um expert em programação gráfica do Apple IIc ou IIe, é essencial saber escrever programas para os controles manuais, como *game paddler, joysticks* e o *mouse*. Neste capítulo, veremos como os controladores funcionam e como são usados em programas gráficos. Serão, também, apresentados programas do tipo "digite e rode", que demonstram como programar os controladores em linguagem assembly. Um dos programas ilustra o uso dos paddles e joysticks, e o outro, o uso do mouse.

A EVOLUÇÃO DOS CONTROLADORES

O primeiro controlador usado nos computadores Apple foi o game paddle (que significa "raquete"). O nome deriva da função, e não da aparência, e este é o motivo pelo qual o paddle não apresenta muita semelhança com outros tipos de raquete. O dispositivo recebeu esse nome pelo fato de ter sido usado para simular as raquetes de pingue-pongue num antigo jogo de fliperama, chamado *Pong*. Eletronicamente, o paddle é um resistor variável; mecanicamente, é constituído, em geral, de um botão rotativo montado em uma base retangular.

Os paddles apresentam-se sempre em pares e são conectados ao mesmo conector de entrada e saída (I/O port).

O Apple IIe tem dois tipos de conectores nos quais podem ser ligados os game paddles: um deles é um soquete DIP de 16 pinos, localizado na própria placa de circuito impresso, onde se vê a inscrição "GAME I/O"; o outro é um conector tipo D, miniatura, no painel traseiro do computador. Ambos os conectores podem também ser usados para instalação de joysticks.

O Apple IIc apresenta somente um conector – tipo D, no painel traseiro – para a conexão de paddles, joystick ou mouse.

COMPARAÇÃO DE PADDLES, JOYSTICK E MOUSE

Como um paddle é apenas um dispositivo unidirecional, pode desenhar uma linha ou mover um objeto em uma única direção, na tela. Já o joystick pode ser movimentado em qualquer direção, fazendo o mesmo com a linha ou objeto na tela. Entretanto, normalmente não oferece tanto controle sobre o posicionamento quanto faz um paddle.

O mouse, quando adequadamente programado, reúne as vantagens dos paddles e do joystick, permitindo mover um objeto em qualquer direção, e de forma bem precisa.

COMO FUNCIONAM OS PADDLES

Vimos que um game paddle é um resistor variável controlado por um botão rotativo. Como os paddles são usualmente conectados aos pares, o Apple IIc/IIe possui duas entradas analógicas. Antes que um programa possa ler essas entradas, deve ativar um circuito de sincronização a elas conectado. Esse circuito pode ser ativado pelo acesso a um *soft switch* situado no endereço de memória $C070. Ao ser ativado, os bits superiores de quatro outros registros – $C064 a $C067 – são ativados. Dois desses registros ($C064 e $C065) correspondem aos dois paddles, numerados como Paddle 0 e Paddle 1.

Se não há paddles ou outros dispositivos conectados aos soquetes de jogos, quando o *soft switch* $C070 é lido, os bits superiores dos registros $C064 a $C067 podem permanecer ativados (em 1) indefinidamente. Se existem, porém, dispositivos ali conectados, os bits superiores de endereços $C064 a $C067 irão, após um período que pode variar entre uma fração de milissegundo e cerca de três milissegundos, mudar de volta para 0, e assim permanecerão.

O tempo exato, necessário para essa mudança dos bits, depende da resistência aplicada ao circuito, que corresponde ao registro de memória afetado. Assim, se o Game Paddle 0 está totalmente para a esquerda, quando o *soft switch* $C070 é acessado, não demorará muito para que o bit superior de $C064 caia de 1 para 0.

Se o Paddle 0 estiver numa posição mais alta, a mudança demorará mais. Da mesma forma, se o Paddle 1 estiver para a esquerda, o bit superior de $C065 mudará

rapidamente de 1 para 0, e demorará mais se o paddle for girado para a direita.

Para ler a condição dos Paddles 0 e 1, um programa deve primeiramente acessar o *soft switch* $C070. A seguir, deve estabelecer um *loop* de tempo, e usá-lo para determinar qual foi o tempo necessário para os bits superiores dos registros $C064 e $C065 caírem de 1 para 0.

Este procedimento representaria um trabalho exaustivo para o programador. Felizmente, os Apple IIc e IIe possuem uma rotina interna que realiza grande parte do trabalho que envolve a leitura do par de game paddles. Essa rotina é chamada PREAD, e seu endereço de início é $FDB1E.

Para usar a rotina PREAD, você tem de armazenar um número de 0 a 3 no registro X do microprocessador. Se estiver usando a rotina PREAD para ler um par de game paddles, podemos armazenar tanto um 0 quanto um 1 no registro X, dependendo do paddle que desejar ler.

Após isso, pode chamar a rotina PREAD efetuando um JSR ao endereço de memória $FB1E. Quando o computador retornar da sub-rotina, um número entre $00 e $FF será armazenado no registro Y do microprocessador. Esse número reflete a posição do paddle.

Como se pode notar, o tempo é crítico, quando se escreve este tipo de programa. E existe ainda uma outra complicação, neste caso particular de programação: pelo fato de poder haver uma demora de até três milissegundos para que os bits superiores dos registros $C064 a $C067 caiam de 1 para 0, torna-se necessária uma espera de, no mínimo, três milissegundos entre a leitura do registro $C064 e a do registro $C065.

Em outras palavras, após haver lido um dos paddles, você deverá aguardar pelo menos três milissegundos antes de tentar ler o outro paddle. Esta situação existe tanto no caso de se executar um programa escrito pelo usuário, para leitura dos paddles, como quando se utiliza a rotina PREAD.

COMO FUNCIONAM OS JOYSTICKS

Os game paddles não vêm sendo muito utilizados em programas Apple, mas os *joysticks*, ou controladores orbitais – descendentes dos game paddles – são ainda bastante populares.

Existem muitas semelhanças entre os programas escritos para paddles e os escritos para joystick. De fato, muitos programas originalmente escritos para paddles funcionam com joystick, e vice-versa.

Como já mencionamos, joysticks e paddles podem ser ligados aos mesmos conectores I/O do Apple IIc e do Apple IIe. Programas projetados para ser usados com joystick são escritos de forma idêntica aos destinados ao uso de paddles. No caso da utilização de joystick, o registro $C064 é usado para ler o movimento da esquerda para a direita, e o $C065 para ler o movimento de cima para baixo, ou seja: o eixo horizontal do joystick é tratado como Paddle 0 e o vertical como Paddle 1 (embora existam alguns programas que invertem esses eixos).

Para ler o joystick em um programa em linguagem assembly, você pode escrever sua própria rotina (usando os registros $C070, $C064 e $C065) ou usar a rotina residente PREAD. Não importa a escolha; é necessário muito cuidado com o tempo. Você pode ler a posição em um mesmo eixo, repetidamente, tão rapidamente quanto o desejar. Porém, deve sempre programar um atraso de, pelo menos, três milissegundos entre a leitura de um eixo e a do outro.

Dois outros endereços de memória freqüentemente usados em programas para joystick são $C061, algumas vezes chamado de Switch Input 0, e $C062, Switch Input 1. Estes endereços podem ser usados para ler os dois botões que existem no controlador joystick ou as teclas MAÇÃ FECHADA e MAÇÃ ABERTA, do teclado do Apple IIc/IIe.

Quando o Botão 0 do joystick ou a tecla MAÇÃ ABERTA são pressionados, o bit 7 do $C061 vai de 1 para 0. Quando o Botão 1 ou a MAÇÃ FECHADA são pressionados, o bit 7 de $C062 cai de 1 para 0. Assim, o estado dos botões do joystick (ou das teclas MAÇÃ ABERTA e MAÇÃ FECHADA) pode ser verificado testando-se o bit 7 (bit de sinal) dos registros de memória $C061 e $C062, para determinar se o número, nessas posições, é maior que 127.

USANDO UM JOYSTICK EM UM PROGRAMA GRÁFICO

Antes de digitar, compilar e rodar um programa em linguagem assembly que ilustra a utilização do joystick em um programa gráfico de baixa resolução, vamos atentar para alguns fatos, a respeito de gráficos de baixa resolução, que não foram abordados no Capítulo 12.

Vimos, no Capítulo 12, que gráficos de baixa resolução constituem um modo gráfico de vídeo, que pode gerar até 16 cores em uma tela de 40 *pixels* de largura por 48 de altura. Você deve também se lembrar de que as coordenadas de colunas no mapa de tela de baixa resolução eram dispostas consecutivamente, mas não o eram as coordenadas de linha.

Como as coordenadas de linhas usadas em todos os mapas de tela dos computadores Apple Série II, as coordenadas de linha em gráficos de baixa resolução são dispostas em três grupos de oito linhas intercalados.

Para escrever programas em gráficos de alta resolução, é conveniente compreender este sistema de mapeamento, detalhadamente, e isto será objeto do Capítulo 14. Entretanto, é possível escrever programas em assembly usando gráficos de baixa resolução, sem envolver-se com os meandros do mapeamento de tela. Os projetistas da Apple providenciaram uma série de rotinas que aliviam grande parcela do esforço na programação em gráficos de baixa resolução:

- *CLRSCR* é uma sub-rotina que limpa a tela de baixa resolução, trazendo-a totalmente à cor preta. A rotina CLRSCR inicia no endereço $F832. Para chamá-la, escreva um comando como este:

  ```
  JSR $F832
  ```

  Se o computador se encontra no modo de baixa resolução, quando este comando é emitido, a tela será limpa. Se o computador estiver no modo de texto em 40 colunas, a tela será preenchida com caracteres "@", porque o código ASCII desse caractere é $00 – o mesmo número usado para a cor preta no modo gráfico de baixa resolução.

- *SETCOL*, uma rotina que inicia no endereço $F864, estabelece a cor que será usada nas operações de marcação dos pontos na tela (plotting). Para usar a rotina SETCOL, carregue o acumulador com o código numérico da cor desejada e efetue um JSR a $F864. (Os códigos para as 16 cores foram apresentados no Capítulo 12.)

- *PLOT*, uma sub-rotina que inicia em $F800, permite marcar (plot) um *pixel*, em qualquer uma das 16 cores, em qualquer posição na tela de baixa resolução. Antes de se chamar a rotina PLOT, uma coordenada de linha, de 0 a 39, deve ser colocada no registro Y, e uma coordenada de coluna, de 0 a 47, deve ser colocada no acumulador. Um único *pixel* será, então, apresentado na tela, na posição especificada. A cor dele pode ser definida pelo uso da rotina SETCOL, antes da chamada da rotina PLOT. As coordenadas de coluna e linha usadas pela rotina PLOT seguem em ordem consecutiva, sem o sistema de intercalamento empregado pelas rotinas do tipo "faça você mesmo".

SKETCHER: UM PROGRAMA EM BAIXA RESOLUÇÃO PARA JOYSTICK

O Programa 13.1 é um programa em linguagem assembly chamado SKETCHER, escrito com o Merlin Pro assembler, mas que também pode ser escrito e compilado com o Apple ProDOS Assembler. Utiliza muitas das técnicas de programação que foram descritas neste capítulo, e outras que serão descritas mais adiante, ainda neste capítulo. Você precisará de um joystick, é claro, para rodar o programa. Antes de rodá-lo, examine-o, linha por linha.

Programa 13.1
O PROGRAMA SKETCHER

```
 1 *
 2 * SKETCHER
 3 *
 4  ORG $8000
 5 *
 6  * ENDERECOS DOS SOFT SWITCHES
 7 *
 8 TEXTOFF EQU $C050
 9 HIRESOFF EQU $C056
10 CARDOFF EQU $C00C
11 MIXEDOFF EQU $C052
12 PAGE2OFF EQU $C054
13 TIMER EQU $C070
14 SWITCH1 EQU $C061
15 SWITCH2 EQU $C062
16 *
17  * ENDERECOS DAS ROTINAS DE FIRMWARE
18 *
19 CLRSCR EQU $F832
20 SETCOL EQU $F864
21 PLOT EQU $F800
22 PREAD EQU $FB1E
23 *
24  * DEFINIR CONSTANTES
25 *
26 PRODL EQU $0300
27 PRODH EQU PRODL+1
28 MPRL EQU PRODH+1
```

```
29 MPRH EQU MPRL+1
30 MPDL EQU MPRH+1
31 MPDH EQU MPDL+1
32 XCOORD EQU MPDH+1
33 YCOORD EQU XCOORD+1
34 XAXIS EQU YCOORD+1
35 YAXIS EQU XAXIS+1
36 *
37  JMP START
38 *
39 MULT16 LDA #0
40  STA PRODL
41  STA PRODH
42  LDX #16
43 SHIFT ASL PRODL
44  ROL PRODH
45  ASL MPRL
46  ROL MPRH
47  BCC NOADD
48  CLC
49  LDA MPDL
50  ADC PRODL
51  STA PRODL
52  LDA MPDH
53  ADC PRODH
54  STA PRODH
55 NOADD DEX
56  BNE SHIFT
57  RTS
58 *
59  * ROTINA DE LOOP DE ATRASO
60 *
61 WAIT LDX #$FF
62 LOOP DEX
63  BNE LOOP
64  RTS
65 *
66  * O PROGRAMA PRINCIPAL INICIA AQUI
67 *
68  * COLOCAR O COMPUTADOR NO MODO GRAFICO DE BAIXA RESOLUCAO
69 *
70 START STA TEXTOFF
71  STA HIRESOFF
72  STA CARDOFF
73  STA MIXEDOFF
74  STA PAGE2OFF
75 *
76  * LIMPAR TELA
77 *
78  JSR CLRSCR
```



```
79 *
80  * ESTABELECER A COR DO PONTO COMO 'ROSA'
81 *
82  STA TIMER
83  * LDA # 11; ROSA
84  JSR SETCOL
85 *
86  * LER COORDENADAS DO JOYSTICK
87 *
88 RDSTICK EQU *
89  JSR WAIT
90  JSR WAIT
91  LDX #0
92  JSR PREAD
93  STY XAXIS
94 *
95  JSR WAIT
96  JSR WAIT
97  JSR WAIT
98  JSR WAIT
99  LDX #1
100  JSR PREAD
101  STY YAXIS
102 *
103  * VERIFICAR CONDICAO DOS BOTOES DO JOYSTICK
104 *
105  LDA SWITCH2
106  BMI START
107  LDA SWITCH1
108  BMI SKIP
109  LDA #0 ;BLACK
110  JSR SETCOL
111  JSR ERASE
112  LDA #11 ;PINK
113  JSR SETCOL
114 *
115  * MARCAR PONTO NA TELA
116 *
117 SKIP LDA #0
118  STA MPDH
119  STA MPRH
120  LDA XAXIS
121  STA MPDL
122  LDA #40
123  STA MPRL
124  JSR MULT16
125  LDA PRODH
126  STA XCOORD
127 *
128  LDA #0
```

```
129   STA MPDH
130   STA MPRH
131   LDA YAXIS
132   STA MPDL
133   LDA #48
134   STA MPRL
135   JSR MULT16
136   LDA PRODH
137   STA YCOORD
138 *
139   LDA YCOORD
140   LDY XCOORD
141   JSR PLOT
142   JMP RDSTICK
143 ERASE LDA YCOORD
144   LDY XCOORD
145   JSR PLOT
146   RTS
```

COMO O PROGRAMA SKETCHER FUNCIONA

O programa inicia na linha 37, com uma instrução JMP que dirige o programa para a linha 70. A partir dessa linha, cinco *soft switches* são ativados. Estes desativam o modo de alta resolução, o firmware de 80 colunas, o modo "mixto" (de forma que não haverá texto na porção inferior da tela) e a memória auxiliar. A seguir, na linha 78, a tela de baixa resolução é trazida à cor preta, com o comando JSR CLRSCR.

Na linha 82, o *soft switch* em $C070 é ativado, iniciando uma contagem regressiva de três milissegundos, de forma que as entradas de game paddle (ou joystick) em $C064 e $C065 possam ser lidas. Depois, o comando JSR SETCOL é usado para estabelecer a cor rosa para o *pixel* de baixa resolução.

O coração do programa SKETCHER é o segmento que se estende da linha 88 à 101. Durante esta seqüência, uma sub-rotina denominada WAIT (que inicia na linha 61) é chamada diversas vezes, para ativar o atraso de três milissegundos que precisa ser inserido entre a leitura de um e outro eixo do joystick. Os demais comandos da seqüência são claros.

Nas linhas de 91 a 93, o registro X é carregado com um 0, e a rotina do monitor PREAD é usada para ler o eixo X do joystick. Nas linhas de 99 a 101, o registro X é carregado com um 1, e a rotina PREAD lê a condição do eixo Y do joystick. Quando este segmento de código termina, a posição do eixo X do joystick (expressa por um número

variando de $00 a $FF) encontrar-se-á no registro de memória intitulado XAXIS, e a posição do eixo Y (expressa da mesma forma) estará no registro de memória YAXIS.

Nas linhas de 105 a 113, o estado dos botões "gatilho" do joystick é verificado, para saber se algum dos botões está sendo pressionado. Se o botão lateral (chamado de Switch 1 neste programa) estiver sendo pressionado, o cursor desenhará uma linha, conforme for se movimentando pela tela. Se o botão superior do joystick (Switch 2) for o pressionado, a tela será limpa. E caso não seja pressionado nenhum dos botões, o cursor movimentar-se-á através da tela, simplesmente, sem desenhar uma linha.

As linhas de 117 a 146 movem o cursor e – se o Switch 2 estiver pressionado – desenham linhas na tela. Nesta seqüência de código, uma rotina de multiplicação em 16 bits é usada para traduzir os números armazenados em XAXIS e YAXIS em duas coordenadas de tela válidas: uma coordenada horizontal (ou X), de 0 a 39, e uma vertical (ou Y), de 0 a 47.

Nas linhas de 117 a 137, uma interessante "rotina" é usada para efetuar aquela conversão. Primeiramente, o valor armazenado em XAXIS é multiplicado por 40, e o byte de ordem superior do resultado é armazenado em um registro de memória chamado de XCOORD.

A seguir, o valor armazenado em YAXIS é multiplicado por 48. Finalmente, o byte de ordem superior do resultado é armazenado no registro de memória chamado de YCOORD. E as duas coordenadas são então usadas para posicionar o cursor na tela.

Isto pode soar como uma forma estranha de cálculo de coordenadas, mas faz bastante sentido nesta aplicação. Extrair o byte superior de um número de 16 bits é como dividir o número por 256. Quando o valor armazenado em XAXIS (um número de $00 a $FF) é multiplicado por 40, e o resultado dividido por 256, XAXIS é convertido em um número que varia de 0 a 39. Esse número – uma coordenada de tela válida – pode, então, ser armazenado em XCOORD.

Nas linhas de 116 a 125, quando o valor em YAXIS é multiplicado por 48 e o resultado é dividido por 256, o valor é convertido em uma coordenada válida, entre 0 e 47, e pode então ser armazenado em YCOORD.

A rotina de marcação (plotting) do programa SKETCHER é o pequeno segmento de código que se estende da linha 139 à linha 146, no qual o valor de YCOORD é armazenado no acumulador, o valor de XCOORD é armazenado no registro Y, e é chamada a sub-rotina PLOT. E, na linha 142, o programa pula para a linha chamada RDSTICK, e outra seqüência de leitura e marcação se inicia.

DESVANTAGENS DOS JOYSTICKS

Ao rodar o programa SKETCHER, você, provavelmente, notará que ele apresenta pequenas deficiências. Por exemplo, se o seu joystick é do tipo autocentrante (e a maioria dos joysticks o é), o cursor sempre retorna ao centro da tela, quando se solta o joystick. Alguns modelos possuem uma característica de desativamento da autocentragem e eliminam este problema. Se o seu joystick não possui esse dispositivo, sempre voltará ao centro, como é o caso do modelo comercializado pelo Apple para o IIc e o IIe.

Se você movimentar rapidamente o joystick, notará outro problema do programa: falhas nas linhas desenhadas na tela, devendo-se este fato aos inevitáveis atrasos do programa, especialmente o atraso de três milissegundos que precisa ser inserido entre a leitura de um e de outro eixo do joystick.

Os problemas de autocentragem e das falhas nas linhas podem ser resolvidos de diversas maneiras. Uma forma de solucionar ambos os problemas simultaneamente é simplificar a forma pela qual a posição do joystick é lida. Em lugar de tentar o acompanhamento exato da posição a todo instante, pode-se usar um programa que simplesmente lê a direção na qual o joystick está sendo mantido. Como mostra o Programa 13.2, tal programa pode ser escrito facilmente, mesmo em BASIC.

Programa 13.2
JOYSTICK.BAS
Um Programa em Baixa Resolução para Joystick

```
10  REM  **** JOYSTICK.BAS ****
20  REM **** PROGRAMA EM BAIXA RESOLUCAO PARA JOYSTICK ****
30  GR : POKE - 16302,0: CALL - 1998 : COLOR = 11 : REM DESLIGAR MODO MIXTO E LIMPAR
    JANELA DE TEXTO
40  XM = PDL (0): X = INT (XM/6.4) : YM = PDL (1) : Y = INT (YM/5.3): REM CONVERTER
    LEITURAS DO JOYSTICK EM COORDENADAS DE LINHAS E COLUNAS EM BAIXA
    RESOLUCAO, E COLOCAR UM PIXEL NO CENTRO DA TELA
50  IF PEEK (49250) > 127 THEN 30 : REM SE O BOTAO 1 ESTIVER PRESSIONADO, REINICIAR
60  IF PEEK (49249) > 127 THEN 80 : REM SE O BOTAO 0 ESTIVER PRESSIONADO, DESENHAR
    LINHA
70  COLOR = 0 : PLOT TX, TY : COLOR = 11 : REM DO CONTRARIO, APAGAR E REMARCAR

75  REM **** ROTINAS DE MARCACAO INICIAM AQUI ****
80  PLOT X, Y : TX = X : TY = Y : REM TX E TY SAO USADOS PARA ARMAZENAMENTO
    TEMPORARIO DAS COORDENADAS X E Y
90  GOSUB 190: IF XP > XM - 5 THEN 120
100 X = X - 1: IF X < 0 THEN X = 0
110  GOTO 140
120  IF XP < XM + 5 THEN 140
```



```
130 X = X + 1: IF X > 39 THEN X = 39
140  IF YP > YM - 5 THEN 160
150 Y = Y - 1: IF Y < 0 THEN Y = 0
160  IF YP < YM + 5 THEN 50
170 Y = Y + 1: IF Y > 47 THEN Y = 47
180  GOTO 50
185 REM **** SUB-ROTINA PARA LEITURA DO JOYSTICK****
190 XP =  PDL (0):YP =  PDL (1): RETURN
```

O programa JOYSTICK.BAS funciona de forma bastante similar ao programa em assembly, SKETCHER, apresentado no Programa 13.1. Entretanto, se digitarmos e rodarmos o JOYSTICK.BAS, notaremos que ele não apresenta os problemas de falhas e autocentragem.

Infelizmente, estas vantagens têm um preço: o programa roda mais lentamente e oferece ao usuário consideravelmente menos controle sobre a posição do cursor. Com o JOYSTICK.BAS, não podemos dirigir o cursor livremente na tela; podemos, somente, movê-lo em oito direções: para cima, para baixo, para a esquerda, para a direita e nas diagonais. E não temos controle sobre a velocidade com que o cursor se movimenta.

Podemos tornar o programa mais rápido, é claro, se o convertermos para linguagem assembly. Mas isto não representará muito, com relação à melhoria de controle do cursor.

Em algumas aplicações, esta limitação de controle sobre a velocidade e o posicionamento do cursor não é importante. Nos jogos tipo "fliperama", por exemplo, raramente é necessário – e, por vezes, nem é desejável – ter absoluto controle sobre a direção e a velocidade do cursor. Já em outras aplicações, como no caso dos programas para desenho, é importante um elevado grau de controle.

O MOUSE DA APPLE

O mouse, como já indicamos neste capítulo, combina as vantagens do joystick com a precisão do paddle. O programa 13-3, intitulado MOUSKETCH, utiliza o mouse para desenhar figuras em baixa resolução, trabalhando de forma similar à do programa SKETCHER.

O programa usa o mouse para movimentar um cursor intermitente através da tela; e se o botão do mouse for pressionado, o cursor desenhará uma linha ao movimentar-se. Existe somente um botão no mouse da Apple, de forma que ele não

poderá ser utilizado para limpar a tela. Entretanto, pressionando-se as teclas MAÇÃ FECHADA ou MAÇÃ ABERTA, a tela será limpa e o programa reiniciará.

Se você tem um Apple IIe, precisará instalar o cartão para mouse no Conector Auxiliar 4, para que o programa opere da forma como está escrito. Se o cartão for instalado em outro conector, será necessário modificar o programa, conforme instruções fornecidas mais adiante, ainda neste capítulo.

Ao compilar e rodar o programa, notaremos que ele apresenta duas grandes vantagens, com relação ao programa SKETCHER. A primeira, devida ao fato do mouse não ser um dispositivo de autocentralização, é que o cursor não retorna ao centro da tela quando soltamos o mouse. A segunda (e, provavelmente, a mais importante) vantagem é que o programa MOUSKETCH praticamente não deixa falhas na linha desenhada, não importando com que velocidade seja deslocado o cursor na tela.

Programa 13.3
O PROGRAMA MOUSKETCH

```
 1 *
 2 * MOUSKETCH
 3 *
 4  ORG $8000
 5 *
 6  JMP GO
 7 *
 8 SLOT EQU $FB
 9 XPSN EQU $3B8
10 YPSN EQU $438
11 BUTTON EQU $6B8
12 *
13 * PONTOS DE DESLOCAMENTO DA ENTRADA DO MOUSE
14 *
15 SETMS EQU $12
16 READMS EQU $14
17 INITMS EQU $19
18 CLAMPMS EQU $17
19 *
20 * ENDEREÇOS DOS SOFT SWITCHES
21 *
22 TEXTOFF EQU $C050
23 HIRESOFF EQU $C056
24 FWOFF EQU $C00C
25 MIXEDOFF EQU $C052
26 PAGE2OFF EQU $C054
27 *
28 * ROTINAS DE FIRMWARE DO MOUSE
29 *
```



```
30 CLRSCR EQU $F832
31 SETCOL EQU $F864
32 PLOT EQU $F800
33 *
34 * DEFINIR CONSTANTES
35 *
36 PRODL EQU $0300
37 PRODH EQU PRODL+1
38 MPRL EQU PRODH+1
39 MPRH EQU MPRL+1
40 MPDL EQU MPRH+1
41 MPDH EQU MPDL+1
42 XCOORD EQU MPDH+1
43 YCOORD EQU XCOORD+1
44 XAXIS EQU YCOORD+1
45 YAXIS EQU XAXIS+1
46 *
47 * BYTES DE INDICACAO (SIGNATURE BYTES)
48 *
49 CN EQU $C4
50 NO EQU $40
51 *
52 * ROTINA AUTO-MODIFICAVEL
53 *
54 SETFW JMP $0000
55 *
56 * O PROGRAMA PRINCIPAL INICIA AQUI
57 *
58 GO EQU *
59 JSR INIT ; INICIALIZAR TELA
60  LDA #$00
61  STA SLOT
62  LDA #CN
63  STA SLOT+1
64  LDY #INITMS
65  JSR CALLFW
66  JSR CLAMP
67  LDY #SETMS
68 LDA#$01 ; ATIVAR MODO PASSIVO
69  JSR CALLFW
70 *
71 * 'LOOP' PRINCIPAL
72 *
73 DOIT EQU *
74  LDY #READMS
75  JSR CALLFW
76  JSR DRAW
77  JMP DOIT
78 *
79 * SUB-ROTINAS INICIAM AQUI
```

```
 80 *
 81 CALLFW EQU *
 82  PHA
 83  LDA (SLOT),Y
 84  LDX #CN
 85  LDY #NO
 86  STA SETFW+1
 87  STX SETFW+2
 88  PLA
 89  JSR SETFW
 90  RTS
 91 *
 92 * DESENHAR PONTO NA TELA
 93 *
 94 DRAW EQU *
 95 *
 96 * VERIFICAR BOTAO DO MOUSE E TECLAS 'MACA'
 97 *
 98  LDA $C061
 99  BMI GO
100  LDA $C062
101  BMI BEGIN
102  LDX SETFW+2
103  LDA BUTTON,X
104  BMI LEAP
105  LDA #0 ;BLACK
106  JSR SETCOL
107  JSR PLOTIT
108  LDA #11 ;PINK
109  JSR SETCOL
110 *
111 * AGORA DESENHAR PONTO
112 *
113 LEAP LDX SETFW+2
114  LDA XPSN,X
115  STA XAXIS
116  LDA YPSN,X
117  STA YAXIS
118 *
119  LDA #0
120  STA MPDH
121  STA MPRH
122  LDA XAXIS
123  STA MPDL
124  LDA #40
125  STA MPRL
126  JSR MULT16
127  LDA PRODH
128  STA XCOORD
129 *
```



```
130  LDA #0
131  STA MPDH
132  STA MPRH
133  LDA YAXIS
134  STA MPDL
135  LDA #48
136  STA MPRL
137  JSR MULT16
138  LDA PRODH
139  STA YCOORD
140 *
141 PLOTIT LDA YCOORD
142  LDY XCOORD
143  JSR PLOT
144  RTS
145 *
146 CLAMP EQU *
147  LDA #$00
148  STA $478
149  STA $578
150  STA $5F8
151  LDA #$FF
152  STA $4F8
153  LDY #CLAMPMS
154  LDA #0
155  JSR CALLFW
156  LDY #CLAMPMS
157  LDA #1
158  JSR CALLFW
159  RTS
160 *
161 MULT16 LDA #0
162  STA PRODL
163  STA PRODH
164  LDX #16
165 SHIFT ASL PRODL
166  ROL PRODH
167  ASL MPRL
168  ROL MPRH
169  BCC NOADD
170  CLC
171  LDA MPDL
172  ADC PRODL
173  STA PRODL
174  LDA MPDH
175  ADC PRODH
176  STA PRODH
177 NOADD DEX
178  BNE SHIFT
179  RTS
```

```
180 *
181 * COLOCAR O COMPUTADOR EM MODO BAIXA RESOLUCAO
182 *
183 INIT  STA TEXTOFF
184   STA HIRESOFF
185   STA FWOFF
186   STA MIXEDOFF
187   STA PAGE2OFF
188 *
189 * LIMPAR TELA
190 *
191   JSR CLRSCR
192 *
193 * ESTABELECER COR DO PONTO
194 *
195   LDA #11 ;PINK
196   JSR SETCOL
197   RTS
```

OBSERVANDO MELHOR O APPLE MOUSE

O programa MOUSKETCH começa na linha 6, com um salto à linha 58, o início real do programa. Na linha 59, a sub-rotina INIT é chamada, e a tela de baixa resolução é inicializada, de forma similar à do programa SKETCHER. A seguir, nas linhas de 59 a 69, o firmware do mouse é inicializado.

Para compreender este processo de inicialização, é interessante conhecer algo sobre a operação e o projeto do mouse. Trata-se de um dispositivo "inteligente"; ele é acompanhado por um firmware complexo, responsável por essa característica, que diferencia o mouse dos dispositivos apenas baseados em resistores, como os paddles e joysticks.

Uma diferença entre o mouse Apple e um simples joystick é o fato do mouse permitir a operação em diferentes modos:

- No *Modo Interrupção por Movimento*, o mouse gera uma interrupção cada vez que é movimentado. Um programa, então, lerá dados gerados pelo mouse apenas quando os dados mudam, em lugar de manter uma constante supervisão para verificar se o mouse foi movido. Este modo é útil nos programas de alto desempenho, onde o tempo de processamento é extremamente valioso. A complexidade do "movement interrupt mode", entretanto, está além do escopo do material introdutório apresentado neste capítulo.

- No *Modo Interrupção por Botão*, a interrupção é gerada quando o botão é pressionado.
- O *Modo Interrupção por Movimento/Botão* combina os dois modos já descritos.
- O *Modo Interrupção por Refrescamento da Tela* (Screen Refresh Interrupt Mode) examina os dados do mouse a cada 1/60 de segundo, durante o intervalo entre duas varreduras de vídeo. Este modo pode eliminar a cintilação (flickering) que, às vezes, surge quando objetos são movidos na tela durante a parte ativa do ciclo de vídeo.
- O *Modo Passivo ou Transparente* usa um sistema de interrupção residente no firmware do mouse, para atualizar automaticamente os dados. O modo transparente é o modo de operação do mouse mais simples, e nele iremos concentrar nossa atenção durante o restante deste capítulo.

SUB-ROTINAS PRÓPRIAS DO MOUSE APPLE

Para estabelecer o modo que será adotado para a operação do mouse, um valor chamado *Byte de Modo* precisa ser carregado no acumulador. A seguir, deve ser efetuado um JSR à sub-rotina SETMOUSE, pertencente ao firmware que opera o mouse. Antes de se chamar a rotina SETMOUSE, entretanto, deverá ser chamada uma rotina preliminar, denominada INITMOUSE.

Além da SETMOUSE e da INITMOUSE, várias outras rotinas importantes estão contidas no firmware do mouse. Para auxiliar os programadores no acesso a essas rotinas, o firmware do mouse contém uma tabela que aponta o endereço de entrada de cada rotina. Num Apple IIc ou IIe equipado com mouse, esta tabela ocupa os endereços de memória $Cn12 a $Cn19, onde a variável n equivale ao número do slot no qual o mouse está instalado. No Apple IIc, este slot é sempre o (Slot)4. No IIe, o slot pode variar, dependendo das expansões instaladas.

A Tabela 13-1 lista os apontadores de endereços que fazem parte do firmware do mouse Apple. Nos programas em assembly, estes apontadores podem ser usados para localizar os bytes inferiores dos endereços de início das rotinas relativas ao mouse. As funções das rotinas serão descritas mais adiante.

Em cada endereço da Tabela 13-1, o byte superior é sempre $Cn, onde n é o número do slot do mouse. Por exemplo, caso o mouse esteja instalado no slot 4, podemos calcular o endereço da rotina INITMOUSE pela adição do conteúdo do endereço $C419

ao valor literal $C400. Podemos determinar o endereço de início da rotina SETMOUSE pela adição do conteúdo do endereço $C412 ao valor $C400.

Conhecido o endereço de início de uma rotina do mouse que desejamos usar, podemos chamar a rotina pelo armazenamento de certos valores nos registros X e Y do microprocessador, efetuando, a seguir, um salto à sub-rotina desejada. Estes são os valores que devem ser armazenados nos registros X e Y antes que qualquer rotina do mouse (exceto a SERVMOUSE) possa ser chamada:

- O valor $Cn (onde n é o número do slot do mouse) deve ser armazenado no registro X.
- O valor $n0 (onde n é o número do slot do mouse) deve ser armazenado no registro Y.

Tabela 13-1 Tabela de Bytes Inferiores dos Endereços das Sub-rotinas do Mouse

Os endereços contêm os bytes inferiores dos endereços de início das rotinas de firmware do mouse:

$Cn12	SETMOUSE
$Cn13	SERVMOUSE
$Cn14	READMOUSE
$Cn15	CLEARMOUSE
$Cn16	POSMOUSE
$Cn17	CLAMPMOUSE
$Cn18	HOMEMOUSE
$Cn19	INITMOUSE

Nas linhas 49 e 50 do programa MOUSKETCH, às duas constantes (CN e N0) são designados os valores $C4 e $40, respectivamente. Se você possui um Apple IIe com o mouse instalado em um slot, que não o de número 4, altere estes valores no programa, de forma a refletirem o slot no qual o mouse está instalado.

CHAMANDO A ROTINA INITMOUSE

Antes de utilizar o mouse, é necessário inicializá-lo, chamando a rotina INITMOUSE. No programa MOUSKETCH, essa rotina é chamada nas linhas de 60 a 65. Nas linhas 59 a 63, o valor $C400 é carregado em um par de registros de 8 bits, na Página Zero, chamados SLOT e SLOT+1. A seguir, usando a tabela de deslocamentos definida nas linhas de 15 a 18, o apontador de deslocamento (offset pointer) para a rotina INITMOUSE – ou seja, o valor 19 – é carregado no registro Y do 6502B/65C02, e uma instrução JSR é usada para pular à sub-rotina CALLFW (abreviatura de *callfirmware*). Esta sub-rotina começa na linha 61.

Quando a rotina CALLFW é chamada, o conteúdo do acumulador é transferido para a pilha, para um armazenamento seguro. Então, na linha 83, tem lugar uma interessante operação. Através de endereçamento indireto indexado, o valor do registro Y – no caso, o número 19 – é somado ao conteúdo do endereço $C400. O número resultante é um endereço de memória que, como se pode constatar na Tabela 13-1, é $C419, que contém o byte inferior do endereço de início da rotina INITMOUSE. Em outras palavras, na linha 83 o byte inferior do endereço de início de INITMOUSE é carregado no acumulador.

A seguir, nas linhas 84 e 85, o registro X é carregado com o valor $Cn e o registro Y, com o valor $n0. Nas linhas 86 e 87, uma técnica complexa de programação, denominada *modificação de endereço* (address modification), é usada para efetuar um JSR à rotina INITMOUSE. Embora difícil de explicar (e talvez mais difícil ainda de entender), a modificação de endereço é freqüentemente usada em programação em linguagem assembly, pois pode conduzir a significativa economia de memória e de tempo de processamento.

Na linha 86, o valor do acumulador – que é o byte inferior do endereço de início da rotina INITMOUSE – é armazenado em um endereço chamado de SETFW+1. Na linha 87, o valor do registro X (o valor $Cn) é armazenado em SETFW+2.

Para achar estes três endereços de memória – SETFW, SETFW+1 e SETFW+2 – volte à linha 54 do programa; eles estão lá.

A linha 54 contém um comando de três bytes: JMP $0000. É uma instrução de três bytes porque contém um byte da instrução e dois do operando. Se você desmembrar os três bytes do comando, verá que, quando o programa MOUSKETCH é compilado, o equivalente em linguagem de máquina do mnemônico *jump* será armazenado no endereço SETFW, e o valor $00 será armazenado em cada um dos dois registros de memória que seguem o endereço SETFW. Em outros termos, os endereços de memória SETFW+1 e SETFW+2 conterão ambos um 0, quando o programa for compilado.

Mas esses zeros, armazenados por ocasião da compilação, são apenas valores "iniciais" (dummy values). Nas linhas 86 e 87, como já vimos, são armazenados novos valores naqueles dois endereços. De fato, no momento em que a rotina CALLFW termina, o endereço de início da rotina INITMS substitui os zeros "iniciais" armazenados em SETFW + 1 e SETFW + 2.

Depois, o valor original do registro A é restaurado (volta da pilha) e uma instrução JSR é usada para pular ao endereço SETFW. Estas duas operações ocorrem nas linhas 88 e 89.

Quando o comando JSR SETFW é encontrado na linha 89, o endereço da *próxima* instrução do programa é colocado na pilha. (Isto acontece sempre que uma instrução JSR é usada em um programa.)

A seguir, o programa primeiramente pula para a linha 54, e depois ao endereço agora armazenado em SETFW + 1 e SETFW + 2 – que é, como já vimos, o endereço de início da rotina INITMS. A rotina INITMS termina como uma instrução RTS que, como todas as instruções RTS, retirará da pilha seu endereço de retorno. Em nosso caso, o endereço recuperado da pilha será o endereço de outra instrução RTS – a da linha 90 do programa. (O endereço desta instrução RTS havia sido colocada na pilha da linha 89.)

Uma vez que o programa encontra seu caminho de volta à instrução RTS da linha 90, essa instrução terminará a rotina CALLFW. O programa, então, retorna para onde estava antes da chamada da sub-rotina CALLFW. Em outras palavras, a instrução RTS da linha 90 levará o programa de volta à linha 66.

Nessa linha, o programa pula para a sub-rotina chamada CLAMP. Esta, que inicia na linha 146, é usada para estabelecer os valores máximo e mínimo que podem ser retornados pelos eixos X e Y do mouse Apple. Quando o mouse é inicializado pela primeira vez, os valores máximo e mínimo, tanto para o eixo X como para o Y, são $00 e $3FF.

No programa MOUSKETCH, a sub-rotina CLAMP é usada pra mudar os valores máximos para $FF – os mesmos valores retornados pelos joysticks e paddles compatíveis com o Apple. No programa MOUSKETCH, os limites para os dados do mouse foram alterados para que as rotinas originalmente escritas para o programa SCKETCHER pudessem também ser usadas no MOUSKETCH. Este procedimento representou uma considerável economia de tempo de programação, quando da escrita do programa MOUSKETCH.

Na rotina CLAMP, uma sub-rotina de firmware do mouse, CLAMPMS, é usada para estabelecer os valores máximo e mínimo das posições X e Y do mouse. Antes de se

chamar a rotina CLAMPMOUSE, o acumulador deve ser carregado com um 0 ou um 1. Se há um 0 no acumulador, quando a CLAMPMOUSE é chamada ela estabelecerá novos valores para as coordenadas X do mouse; se há um 1 no acumulador, a rotina alterará os limites das coordenadas Y do mouse.

Quando a CLAMPMOUSE é executada, os conteúdos de quatro endereços de memória tornam-se as novas fronteiras superior e inferior das coordenadas de tela do mouse. Estes quatro endereços e suas respectivas funções são

$478	Byte inferior do novo limite inferior
$4F8	Byte inferior do novo limite superior
$578	Byte superior do novo limite inferior
$5F8	Byte superior do novo limite superior

Durante a sub-rotina CLAMP, a CLAMPMOUSE é chamada duas vezes – uma para cada coordenada do mouse. Então, o programa retorna para a linha 67, onde a rotina SETMOUSE é chamada. No programa MOUSKETCH, a rotina SETMOUSE é usada para colocar o mouse no modo passivo.

Antes que a rotina SETMOUSE seja chamada, é necessário armazenar um valor no acumulador. A Tabela 13-2 apresenta os Bytes de Modo que podem ser usados com a sub-rotina SETMOUSE.

Tabela 13-2 Bytes de Modo Usados com a Rotina SETMOUSE

$00	Desativa o mouse
$01	Estabelece o modo transparente
$03	Estabelece o modo de interrupção por movimento
$05	Estabelece o modo de interrupção por botão
$07	Estabelece o modo de interrupção por movimento/botão
$08-$0F	Bytes usados para estabelecer os modos de interrupção por refrescamento de tela no Apple IIc

Após a chamada da sub-rotina CLAMP, o programa vai para o *loop* principal, intitulado DOIT, que começa na linha 73. Nesse loop, uma rotina do firmware do mouse, READMOUSE, é usada para ler a condição do dispositivo.

O programa, então, pula para uma sub-rotina de nome PLOTDOT, que inicia na linha 94. Essa rotina trabalha quase que exatamente como a rotina RDSTICK do programa SKETCHER: ela verifica as teclas MAÇÃ FECHADA e MAÇÃ ABERTA, para saber se a tela deve ser limpa, e verifica o botão do mouse, para saber se uma linha deve ser desenhada conforme o cursor se movimenta. Quando a sub-rotina PLOTDOT termina, uma instrução JMP é usada para reiniciar o loop DOIT novamente.

CAPÍTULO 14

GRÁFICOS APPLE

Se você conhece BASIC – mesmo só um pouco – pode escrever programas de grande efeito em gráficos de baixa resolução. Para criar programas de ação rápida e efeito surpreendente em alta resolução, entretanto, é quase indispensável o uso da linguagem assembly.

Neste capítulo você aprenderá os princípios dos gráficos de alta resolução e de dupla alta resolução, e terá oportunidade de digitar, compilar e executar dois programas em alta resolução. O primeiro lhe permitirá digitar caracteres na tela quando o computador estiver no modo alta resolução, e o segundo fará o mesmo com caracteres "garrafais", quatro vezes maiores que os normais, e em cores, caso você tenha um monitor colorido.

Estes dois programas somente farão sentido se você dispuser de um conhecimento fundamental a respeito de como o Apple IIc e o Apple IIe geram apresentações em alta resolução.

A Figura 14-1 é um mapa da tela de Alta Resolução, Página 1, o bloco de RAM mais freqüentemente utilizado como memória de tela nos Apple IIc e IIe. Este bloco de RAM estende-se do endereço de memória $2000 ao $3FFF.

Note que o mapa de tela ilustrado pela Figura 14-1 é bastante similar ao mapa de texto em 40 colunas ilustrado no Capítulo 12: é constituído de 1024 retângulos dispostos em uma grade, com largura de 40 colunas e altura de 24 linhas. O sistema usado é o mesmo, em ambos os mapas, para a localização de retângulos individuais.

Em cada mapa, o primeiro byte de cada linha horizontal é identificado por um endereço de início de dois bytes, e cada coluna vertical é identificada por um deslocamento de um byte.

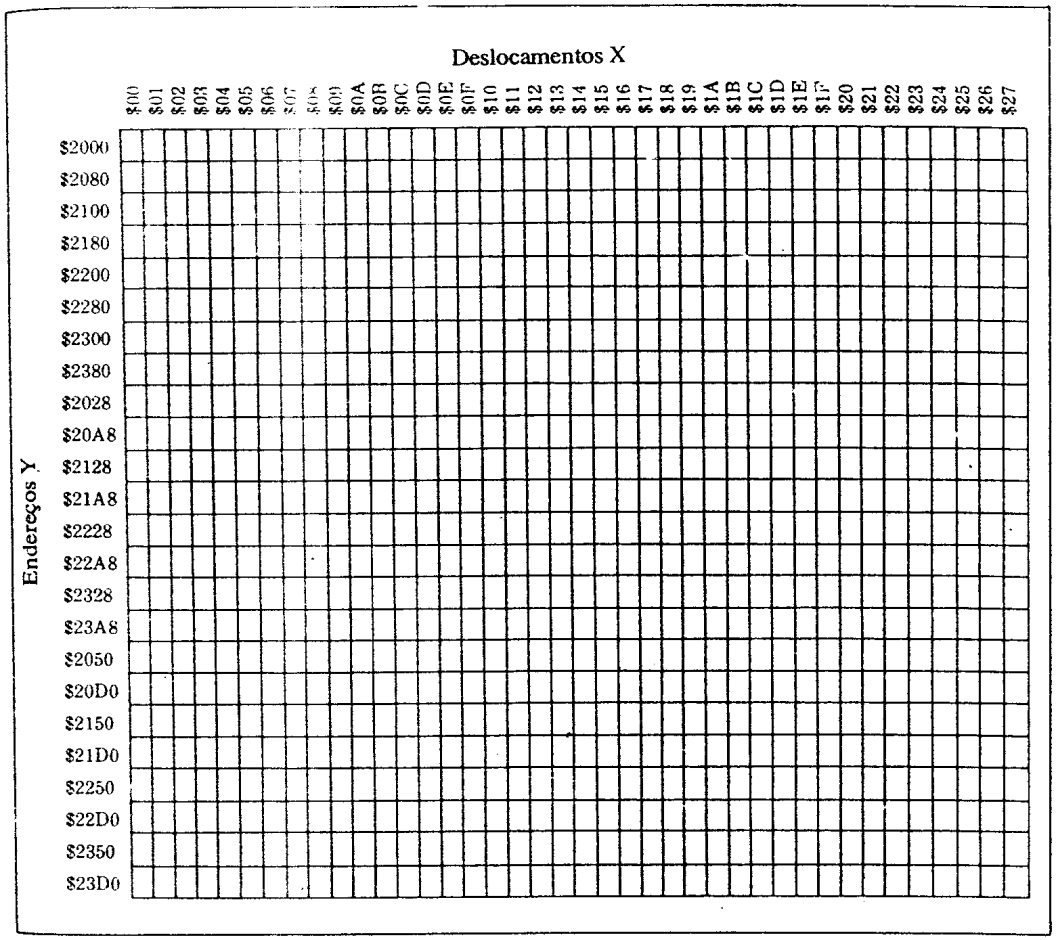


Figura 14-1 Mapa de tela para gráficos de alta resolução.

Para apontar a localização de um dado retângulo, em qualquer um dos mapas, adicionamos o deslocamento de coluna, ou deslocamento X, ao endereço de linha, ou endereço Y. O resultado será o endereço de início do retângulo a ser acessado.

Os endereços de linha listados ao longo da porção esquerda da Figura 14-1 são dispostos exatamente da mesma maneira que os endereços Y no mapa de texto em 40 colunas apresentado na Figura 12-1: em três grupos de oito linhas.

No mapa de alta resolução da Figura 14-1, o primeiro grupo de endereços de linha abrange os endereços de $2000 a $2380, o segundo grupo, os endereços de $2028 a $23A8 e o terceiro, os endereços de $2050 a $23D0.

À primeira vista, a esquematização dos dois mapas parece ser exatamente a mesma. Entretanto, existe uma importante diferença na forma pela qual os dois mapas são interpretados nos programas.

O mapa de texto da Figura 12-1 é exatamente o que parece ser: um mapa de 1024 bytes de dados. No entanto, o mapa da Figura 14-1 representa, na realidade, 8192 bytes de dados – ou $2000 bytes em notação hexadecimal. A próxima secção explicará como 8192 bytes podem ser encaixados em 1024 retângulos.

COMO O APPLE IIc/IIe CRIA APRESENTAÇÕES NA TELA

Quando se deseja que um Apple IIc/IIe apresente um caractere em uma tela de texto em 40 colunas, basta armazenar o código ASCII Apple desse caractere no mapa de tela da Figura 12-1. O computador, então, converterá o código em um determinado padrão de pontos, e apresentará esse padrão na posição apropriada na tela.

A Figura 14-2 ilustra o padrão de pontos gerado e apresentado pelo sistema operacional do Apple IIc/IIe quando a tecla correspondente à letra A é pressionada no teclado.

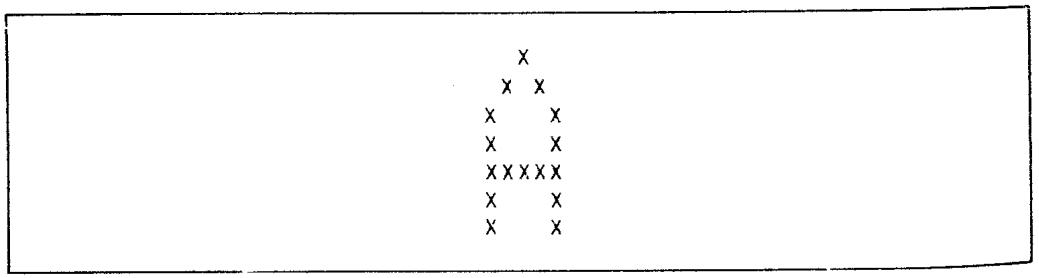


Figura 14-2 Padrão de pontos na tela para a letra A.

Quando o computador se encontra em modo de texto em 40 colunas, cada retângulo do mapa de tela representa um único código de caractere ou apenas um byte de dados. Porém, quando esse byte passa através do circuito gerador de caracteres, é convertido em oito bytes (64 bits) de dados. Estes oito bytes são, então, usados para criar o padrão de pontos na tela.

Mas, quando o computador está em modo de alta resolução, ele gera o que é conhecido como apresentação na tela mapeada em bits. Quando o mapeamento em bits é usado para criar apresentações na tela, o programador tem controle direto sobre cada ponto apresentado na tela. Em lugar de armazenar um byte em uma posição do mapa de tela e legar ao computador a tarefa de converter aquele byte em um padrão de bits, o programador tem de armazenar oito bytes de dados em *cada um* dos 1024 retângulos existentes no mapa de memória. O mapeamento em bits de uma tela de alta resolução, portanto, requer oito vezes 1024 bytes ou 8192 bytes ($2000 bytes em notação hexadecimal).

Como são necessários 8192 bytes para mapear uma tela de alta resolução, e somente existem 1024 células no mapa da Figura 14-1, é óbvio que algum truque deve ser usado. Uma vez que oito linhas de pontos, numa tela de alta resolução, ocupam o mesmo espaço que uma linha de caracteres, o número total de linhas de pontos em uma tela de alta resolução é de oito vezes 24 ou 192.

Para apresentar todas essas linhas em uma tela de alta resolução, o subgrupo representado pelas primeiras linhas de pontos no primeiro grupo de oito linhas do mapa de tela é armazenado nos primeiros 1024 bytes do bloco de memória que está sendo usado para a apresentação em alta resolução.

O subgrupo formado pelas segundas linhas de pontos é armazenado nos 1024 bytes seguintes da memória de tela, e assim por diante, num total de oito vezes 1024 ou 8192 bytes. Em outras palavras, cada bloco de 1024 bytes na página do display de alta resolução contém uma linha de pontos de cada grupo de oito linhas.

Existe uma pequena (e simples) equação que pode ser usada para localizar o endereço de memória de qualquer byte em uma tela de alta resolução. O Programa 14-1 é um programa em BASIC que contém esta equação.

No programa, Q significa "quociente", R significa "resto" e VP representa a posição vertical de um ponto na tela, expressa por um número de linha, de 0 a 192. Quando a equação é resolvida, fornece o valor de YP, que representa "posição Y". Nesta equação, a posição Y é o endereço de início da linha do mapa que contém o byte desejado. Determinado esse valor, pode-se adicionar o deslocamento X do byte ao valor de YP. O resultado é o endereço de memória do byte.

Programa 14-1
FÓRMULA PARA LOCALIZAR UM NÚMERO DE LINHA EM UMA TELA DE ALTA RESOLUÇÃO*

```
10 Q1 = INT (VP / 8):R1 = VP – Q1 * 8
20 Q2 = INT (Q1 / 8):R2 = Q1 – Q2 * 8
30 YP = 8192 + Q2 * 40 + R2 + 128 + R1 * 1024
```

* Adaptado de um programa do livro *Graphically Speaking*, de Mark Pelczarski (Softalk Books, 830 Fourth Avenue, Geneva, Illinois 60134, 1983). Usado com permissão.

Se desejar escrever um programa de gráficos de alta resolução em BASIC, você poderá usar uma equação como a do Programa 14-1, sempre que quiser marcar um ponto na tela. Faz mais sentido, entretanto, usar a equação para estabelecer os 192 endereços Y da tela de alta resolução. O programa poderá, então, localizar os endereços Y de uma linha da tela, a qualquer momento, simplesmente consultando a tabela.

Programa 14-2
PROGRAMA PARA CRIAÇÃO DE UMA TABELA DE ENDEREÇOS Y*

```
10  FOR VP = 0 TO 191
20 Q1  =  INT (VP / 8):R1 = VP – Q1 * 8
30 Q2  =  INT (Q1 / 8):R2 = Q1 – Q2 * 8
40 YP  =  8192 + Q2 * 40 + R2 + 128 + R1 * 1024
50  POKE 28672 + VP, INT (YP / 256)
60  POKE 28864 + VP,YP – INT (YP / 256) * 256
70  NEXT VP
```

* Adaptado de um programa do livro *Graphically Speaking*, de Mark Pelczarski (Softalk Books, 830 Fourth Avenue, Geneva, Illinois 60134, 1983). Usado com permissão.

O Programa 14-2 cria a tabela de endereços Y e a armazena em um segmento de RAM que inicia no endereço de memória $7000 (28672 em notação decimal). O byte inferior de cada endereço Y é armazenado no bloco de memória que começa em $7000, e o byte superior de cada endereço Y é armazenado em um segundo bloco, que começa em $70C0 (28864 em decimal).

Este programa pode parecer um tanto estranho, mas, na realidade, facilita o uso da tabela, uma vez que o deslocamento que localiza o byte superior do endereço Y serve também para localizar o byte inferior.

Quando usar um programa como o Programa 14-2 para criar uma tabela, você poderá armazenar a tabela em disco, como um arquivo binário, com um comando ProDOS como

```
BSAVE TABLE, A28672, L384
```

Como você deve saber, o número após o A, neste comando, é o endereço (address) no qual se deseja armazenar a tabela na memória. O número após o L é o comprimento (length), em bytes, do bloco de dados que se deseja armazenar.

Esta técnica pode ser usada para armazenar programas em linguagem de máquina em disco, da mesma forma que armazenamos dados. E, é lógico, uma vez que um programa, ou tabela de dados, tenham sido armazenados em disco, podem ser facilmente recuperados do disco e incorporados em outros programas em linguagem de máquina.

Maiores detalhes sobre este processo podem ser encontrados no *ProDOS Technical Manual*, da Apple, ou em outros manuais de ProDOS.

Você pode digitar e rodar o Programa 14-2 se desejar, mas isso não é realmente necessário. A mesma rotina, traduzida em linguagem assembly, aparece no próximo programa que examinaremos. Esse programa, chamado PRNTCHRS, é apresentado no Programa 14-3, e foi criado com um Merlin Pro Assembler. Com pequenas modificações, funcionará também quando digitado e rodado usando-se um Apple ProDOS Assembler.

Programa 14-3
O Programa PRNTCHRS

```
 1 *
 2 * PRNTCHRS.S
 3 *
 4  ORG $6FFD
 5 *
 6  JMP START
 7 *
 8 KYBD EQU $C000
 9 STROBE EQU $C010
10 *
11 TEMPLO EQU $06
12 TEMPHI EQU TEMPLO+1
13 CBASLO EQU TEMPHI+1
14 CBASHI EQU CBASLO+1
15 TABPTR EQU $4A
16 *
17 FILVAL EQU $300
18 TABSIZ EQU FILVAL+1
19 QUOT1 EQU TABSIZ+1
20 QUOT2 EQU QUOT1+1
21 REMDR1 EQU QUOT2+1
22 REMDR2 EQU REMDR1+1
23 YLINE EQU REMDR2+1
24 PRODL EQU YLINE+2
25 PRODH EQU PRODL+1
26 MPRL EQU PRODH+1
27 MPRH EQU MPRL+1
28 MPDL EQU MPRH+1
29 MPDH EQU MPDL+1
30 TOTAL EQU MPDH+1
31 XPSN EQU TOTAL+2
32 YPSN EQU XPSN+1
33 *
34 KILOBYTE EQU 1024
35 SCRTOP EQU $2000
36 *
```

```
37 PTRH EQU $8000
38 PTRL EQU $8000
39 *
40 CHRTAB EQU $7000
41 *
42  HEX 2024282C3034383C2024282C3034383C
43  HEX 2125292D3135393D2125292D3135393D
44  HEX 22262A2E32363A3E22262A2E32363A3E
45  HEX 23272B2F33373B3F23272B2F33373B3F
46  HEX 0000000000000000808080808080808080
47  HEX 0000000000000000808080808080808080
48  HEX 0000000000000000808080808080808080
49  HEX 0000000000000000808080808080808080
50  HEX 282828282828282828A8A8A8A8A8A8A8A8
51  HEX 282828282828282828A8A8A8A8A8A8A8A8
52  HEX 282828282828282828A8A8A8A8A8A8A8A8
53  HEX 282828282828282828A8A8A8A8A8A8A8A8
54  HEX 505050505050505050D0D0D0D0D0D0D0D0
55  HEX 505050505050505050D0D0D0D0D0D0D0D0
56  HEX 505050505050505050D0D0D0D0D0D0D0D0
57  HEX 505050505050505050D0D0D0D0D0D0D0D0
58  HEX D000000000000000001C1C1C1C001C1C
59  HEX 00363624360000000123F3F123F3F12
60  HEX 000C3F033F303F0C000027170F3C3A39
61  HEX 00060902042A112E000C0C080C000000
62  HEX 00381C0E0E0E1C38000E1C3838381C0E
63  HEX 00082A1C3E1C2A08000C0C3F3F0C0C00
64  HEX 0000000000001C180C00003E3E000000
65  HEX 0000000000001C1C006070381C0E0703
66  HEX 001E33333333331E003C363330303030
67  HEX 001E3F33380E3F3F001E3F303E303F1E
68  HEX 00383C36333F3030003F3F031F303F1E
69  HEX 001C06031F33331E003F3F30180C0C0C
70  HEX 001E3F331E333F1E001E3F333E303E1E
71  HEX 00001C1C00001C1C00001C1C00001C18
72  HEX 0C70381C0E1C387000003E3E003E3E00
73  HEX 00070E1C381C0E07001E3F33180C00
74  HEX 0C007F7F7F7F7F7F7F1E3F333F3F33
75  HEX 33001F3F331F333F1F001E3F3303333F
76  HEX 1E001F3F3333333F1F003F3F031F033F
77  HEX 3F003F3F031F1F0303001E3F033B333F
78  HEX 1E003333333F3F3333003F3F0C0C0C3F
79  HEX 3F00B0B0B0B0B3BF9E80333B1F0F1F3B
80  HEX 330003030303033F3F0063776B636363
81  HEX 63003333373F3B3333001E3F3333333F
82  HEX 1E001F3F333F1F0303001E3F23232B13
83  HEX 2E001F3F333F1F3B33001E3F031E303F
84  HEX 1E003F3F0C0C0C0C0C0033333333333F
85  HEX 1E0033333333331E0C006363636B6B77
86  HEX 630033331E0C1E33330033333331E0C0C
```



```
87  HEX 0C003F3F180C063F3F001F1F0303031F
88  HEX 1F0003070E1C387060001F1F1818181F
89  HEX 1F000C1E3F0000000000808080808080
90  HEX 80FFFF060C1830000000000001E303E
91  HEX 333E0003031F3333331F0000001E3303
92  HEX 331E0030303E3333333E0000001E331F
93  HEX 031E001C36061F0606060000001E3333
94  HEX 3E301E03031F3333333300000C000C0C
95  HEX 0C0C003000303030333331E030331B0F
96  HEX 1B33000E0C0C0C0C0C1E0000003F5B5B
97  HEX 5B5B0000001F3333333300000001E3333
98  HEX 331E0000001F33331F030300003E3333
99  HEX 3E303000001F330303030000001E031E
100 HEX 301E0006061F0606361C000000333333
101 HEX 333E0000003333331E0C0000006D6D6D
102 HEX 6D7E000000331E0C1E33000000333333
103 HEX 3E301E00003F180C063F001C1E060706
104 HEX 1E1C000C0C0C0C0C0C0C0C0E1E183818
105 HEX 1E0E000000281400000000FFFFFFFF
106 *
107 * ESTABELECER TABELA DE COORDENADAS DE LINHAS
108 *
109 START LDY #0
110 YLOOP CPY #192
111  BCC CONT
112  JMP EXIT
113 *
114 * DIVIDIR Y POR 8
115 *
116 CONT TYA
117  LSR
118  LSR
119  LSR
120  STA QUOT1
121 *
122 * OBTER RESTO DA DIVISAO POR 8
123 *
124  TYA
125  AND #7
126  STA REMDR1
127 *
128 * DIVIDIR QUOT1 POR 8
129 *
130  LDA QUOT1
131  LSR
132  LSR
133  LSR
134  STA QUOT2
135 *
136 * OBTER RESTO
```

```
137 *
138  LDA QUOT1
139  AND #7
140  STA REMDR2
141 *
142 *  CALCULAR BYTE INFERIOR DO ENDEREÇO Y
143 *
144  LDA #0
145  STA MPRH
146  STA MPDH
147  LDA QUOT2
148  STA MPRL
149  LDA #40
150  STA MPDL
151  JSR MULT16
152  LDA PRODL
153  STA TOTAL
154  LDA PRODH
155  STA TOTAL+1
156 *
157  LDA #0
158  STA MPRH
159  STA MPDH
160  LDA REMDR2
161  STA MPRL
162  LDA #128
163  STA MPDL
164  JSR MULT16
165  CLC
166  LDA PRODL
167  ADC TOTAL
168  STA TOTAL
169  LDA PRODH
170  ADC TOTAL+1
171  STA TOTAL+1
172 *
173  LDA #0
174  STA MPRH
175  LDA REMDR1
176  STA MPRL
177  LDA #<KILOBYTE
178  STA MPDL
179  LDA #>KILOBYTE
180  STA MPDH
181  JSR MULT16
182  CLC
183  LDA PRODL
184  ADC TOTAL
185  STA TOTAL
186  LDA PRODH
```



```
187  ADC TOTAL+1
188  STA TOTAL+1
189 *
190  CLC
191  LDA #<SCRTOP
192  ADC TOTAL
193  STA PTRL,Y
194  LDA #>SCRTOP
195  ADC TOTAL+1
196  STA PTRH,Y
197 *
198  INY
199  JMP YLOOP
200 *
201 EXIT EQU *
202 *
203 *  INICIALIZAR APRESENTACAO NA TELA
204 *
205 INIT STA $C050 ;DESATIVAR MODO DE TEXTO
206  STA $C052 DESATIVAR MODO MIXTO
207  STA $C057 ATIVAR MODO ALTA RESOLUCAO
208 *
209 *  LIMPAR TELA
210 *
211  LDA #0
212  STA FILVAL
213  LDA #$00
214  STA TABPTR
215  STA TABSIZ
216  LDA #$20
217  STA TABPTR+1
218  STA TABSIZ+1
219  JSR BLKFIL
220 *
221 *  APRESENTAR CARACTERE NA TELA
222 *
223  LDA #0
224  STA XVALUE
225  STA XPSN
226  STA YVALUE
227  STA YPSN
228 PRINTIT LDA KYBD
229  CMP #$80
230  BCC PRINTIT
231  STA STROBE
232  AND#$7F ;LIMPAR BIT SUPERIOR
233  STA CHAR
234  LDX XPSN
235  STX XVALUE
236  LDY YPSN
```

```
237   STY YVALUE
238   JSR PRNTCHRS
239  *
240  * 'RESET' NAS COORDENADAS DA TELA
241  *
242   LDX XPSN
243   INX
244   CPX #40
245   BCC NEXT
246   CLC
247   LDA YPSN
248   ADC #8
249   CMP #185
250   BCC ITSOK
251   LDA #0
252 ITSOK STA YPSN
253   LDX #0
254 NEXT STX XPSN
255   JMP PRINTIT
256  *
257  * ROTINA DE MULTIPLICACAO EM 16 BITS
258  *
259 MULT16 LDA #0
260   STA PRODL
261   STA PRODH
262   LDX #16
263 SHIFT ASL PRODL
264   ROL PRODH
265   ASL MPRL
266   ROL MPRH
267   BCC NOADD
268   CLC
269   LDA MPDL
270   ADC PRODL
271   STA PRODL
272   LDA MPDH
273   ADC PRODH
274   STA PRODH
275 NOADD DEX
276   BNE SHIFT
277   RTS
278  *
279  * ROTINA PRNTCHRS
280  *
281 PRNTCHRS LDA #<CHRTAB
282   STA CBASLO
283   LDA #>CHRTAB
284   STA CBASHI
285   LDA CHAR
286   LSR
```



```
287   LSR
288   LSR
289   LSR
290   LSR
291   CLC
292   ADC CBASHI
293   STA CBASHI
294   LDA CHAR
295   AND #$1F
296   ASL
297   ASL
298   ASL
299   CLC
300   ADC CBASLO
301   STA CBASLO
302   LDX #0
303 LOOP LDY YVALUE
304   LDA PTRL,Y
305   STA TEMPLO
306   LDA PTRH,Y
307   STA TEMPHI
308   TXA
309   TAY
310   LDA (CBASLO),Y
311   LDY XVALUE
312   STA (TEMPLO),Y
313   INC YVALUE
314   INX
315   CPX #8
316   BNE LOOP
317   RTS
318  *
319  * ROTINA DE PREENCHIMENTO DE BLOCO
320  *
321 BLKFIL LDA FILVAL
322   LDX TABSIZ+1
323   BEQ PARTPG
324   LDY #0
325 FULLPG STA (TABPTR),Y
326   INY
327   BNE FULLPG
328   INC TABPTR+1
329   DEX
330   BNE FULLPG
331 PARTPG LDX TABSIZ
332   BEQ FINI
333   LDY #0
334 PARTLP STA (TABPTR),Y
335   INY
336   DEX
```

```
337  BNE PARTLP
338 FINI RTS
339 *
340 CHAR DFB 0
341 XVALUE DFB 0
342 YVALUE DFB 0
343 *
```

EXAMINANDO O PROGRAMA PRNTCHRS

O programa PRNTCHRS inclui um extenso bloco de números hexadecimais, da linha 42 a 105. Esta secção do programa é, na realidade, uma tabela de dados (bits) que equivale a um jogo de caracteres similar ao produzido pela ROM geradora de caracteres do Apple IIc.

Jogos de caracteres pré-desenhados, como o utilizado neste programa, são incluídos em diversos utilitários gráficos. Por exemplo, o jogo de caracteres do programa PRNTCHRS faz parte do *The Complete Graphics System*, da Penguin Software, e foi aqui utilizado com permissão.

Se você possui um pacote de software gráfico que contém um jogo de caracteres, pode substituir o jogo de caracteres do programa PRNTCHRS. Pode, inclusive, desenhar seu próprio jogo de caracteres, se o desejar.

Caso se utilize um jogo substituto, ele deve ser carregado em RAM a partir do endereço $7000. Se o jogo começar em algum outro endereço, poderá ser copiado num disco, carregado de volta à memória iniciando em $7000, e copiado novamente em disco, como um novo arquivo, com endereço de início em $7000.

Uma vez de posse de um jogo de caracteres iniciando em $7000, você poderá substituir o do programa PRNTCHRS, apagando as linhas de 42 a 105, carregando seu novo jogo de caracteres em RAM, e simplesmente rodando o programa. Ao usar um jogo de caracteres substituto, certifique-se de *não* apagar a linha 40, que diz ao programa onde procurar por um jogo de caracteres na memória.

O programa PRNTCHRS começa na linha 6, com uma instrução JMP que leva o programa à linha 109. Na secção do programa que se inicia ali, e estende-se até a linha 201, uma tabela de endereços Y é criada e armazenada em um bloco de memória que começa em $8000. A rotina que cria essa tabela funciona exatamente como a do Programa 14-2. Embora maior que a rotina do Programa 14-2, a rotina do PRNTCHRS é mais rápida, por ser escrita em linguagem assembly.

Quando o programa termina a criação e o armazenamento da tabela de endereços Y, ele inicializa uma tela gráfica de alta resolução e limpa a Página 1 de Alta Resolução, armazenando em cada byte dela um 0, correspondente à cor preta.

Para limpar a tela, o programa utiliza uma sub-rotina de preenchimento de bloco que inicia na linha 321. Esta rotina é bastante útil, pois permite preencher rapidamente qualquer bloco de RAM com qualquer valor desejado.

Usando uma variável chamada FILVAL (fill value = preencher valor) e técnicas de endereçamento de byte superior e inferior, o trabalho é realizado em duas etapas. Primeiramente, a rotina preenche tantas páginas de 1K (não páginas de tela) quanto possível, com o valor de FILVAL. A seguir, preenche quaisquer páginas parciais remanescentes.

O coração do programa PRNTCHRS é a secção que se estende da linha 223 à linha 228. A primeira coisa que acontece neste bloco de código é que as coordenadas correspondentes ao primeiro caractere a ser apresentado – representadas pelas variáveis XVALUE e YVALUE – são estabelecidas em 0. O primeiro caractere digitado, portanto, será apresentado no canto superior esquerdo da tela.

Após o estabelecimento das coordenadas, um algoritmo de leitura do teclado, comumente utilizado, tem início. Dois importantes registros de memória aparecem neste algoritmo: o endereço $C000, intitulado KYBD, e o endereço $C010, chamado STROBE. KYBD é um endereço de ROM que pode ser verificado para se constatar se uma tecla foi pressionada, e STROBE é um *soft switch* que permite limpar um caractere do teclado, após sua leitura.

Quando é pressionada uma tecla, no teclado do Apple IIc/IIe, vários fatos ocorrem. Primeiro, o código ASCII do caractere digitado é armazenado no acumulador. A seguir, o bit 7 do registro de memória $C000 é ativado. Mantendo o endereço $C000 sob observação, um programa pode saber quando um caractere foi digitado. Tão logo isto ocorra, o código ASCII do caractere pode ser recolhido do acumulador.

No programa PRNTCHRS, o endereço de memória $C000 é verificado nas linhas 228 e 229. Na linha 228, o acumulador é carregado com o valor de $C000; na 229, esse valor é comparado com o valor #$80. Se o conteúdo de $C000 for menor que 80, então o bit 7 do $C000 não foi ativado, o que indica que não foi digitado nenhum caractere. Neste caso, o programa fará um *loop* de volta à linha 228.

Caso o bit 7 de $C000 esteja ativado, indica que um caractere foi digitado, e então o programa limpará o strobe do teclado, acessando o *soft switch* $C010. (Foi aqui utilizada uma operação de gravação, porém, devido à forma de operação dos *soft switches* no

Apple II, uma leitura também limparia o strobe.) Limpando-se o strobe, limpa-se também o teclado, de forma que outra tecla pode então ser lida.

Após uma tecla haver sido pressionada e o strobe limpo, o bit superior do valor presente no acumulador é desativado, e o valor resultante é armazenado em uma variável chamada CHAR. O bit superior do acumulador é desativado porque o jogo de caracteres do programa PRNTCHRS não inclui caracteres invertidos (pretos em fundo branco), intermitentes ou jogo alternativo de caracteres. No sistema ASCII Apple, a tais caracteres são associados números de código ASCII que têm seus bits superiores ativados.

Após o código ASCII de um caractere ter sido armazenado na variável CHAR, o padrão de bits correspondente ao caractere precisa ser encontrado no jogo de caracteres que inicia no endereço $7000, para então o padrão correspondente ao caractere poder ser apresentado na tela.

O processo de procura do padrão de bits do caractere começa na linha 281. Nas linhas de 281 a 283, o endereço de início da tabela do jogo de caracteres – chamada de CHRTAB – é carregado em um par de variáveis, chamadas CBASLO (character base-low) e CBASHI (character base-high).

A seguir, o acumulador é carregado com o código ASCII armazenado na variável CHAR, e uma série de instruções LSR é usada para dividir esse valor por 32. Esta operação é realizada porque a tabela de caracteres ocupa quatro páginas de memória de 1K, e cada página contém dados de bits para 32 caracteres.

Como a tabela é disposta em ordem de códigos ASCII, a divisão do código por 32 indicará a que página o caractere pertence na tabela. Então, como são necessários 8 bits para formar um caractere, o resto, multiplicado por 8, nos fornecerá a localização do caractere naquela página.

Nas linhas de 294 a 301, um belo truque é utilizado para calcular o resto que deve ser multiplicado por 8. Neste segmento do programa, uma operação lógica AND é realizada com o número $1F (decimal 31) e com o código ASCII armazenado em CHAR. Como 31 é 32 menos 1 – ou 0001 1111 em notação binária – o uso de AND, com qualquer número e 31, fornecerá o resto da divisão desse número.

Nas linhas de 229 a 392, o quociente do problema de divisão que acabamos de efetuar é somado ao byte superior de CHRTAB – o endereço de início da tabela de caracteres. O resto da divisão por 32 é somado ao byte inferior de CHRTAB e armazenado em CBASLO. Após a execução destas operações, o byte de ordem superior do endereço do caractere que estamos procurando estará armazenado na variável CBASHI, e o byte inferior estará armazenado na variável CBASLO.

Na secção seguinte do programa PRNTCHRS, as variáveis XPSN e YPSN (posição X e posição Y) são usadas para representar as coordenadas do último caractere apresentado na tela, enquanto XVALUE e YVALUE representam o deslocamento X do próximo caractere a ser apresentado. Duas constantes, PTRL e PTRH (pointer-low e pointer-high), contêm os endereços de início das tabelas de bytes superior e inferior de Y armazenadas nos endereços $8000 e $80C0. Mais duas variáveis, TEMPLO e TEMPHI, contêm os endereços de início dos dados de bits que serão usados para apresentar o caractere desejado na tela.

O restante do programa PRNTCHRS usa endereçamento indireto, juntamente com *loops* que incrementam e decrementam registros, para apresentar os padrões de bits na tela. Usando estas técnicas, o programa recolhe bytes consecutivos de dados de caracteres, na tabela de caracteres em $7000. Então, usando o mesmo deslocamento X, e diferentes endereços Y, o programa armazena estes oito bytes em uma pilha, que forma o casejado na tela.

APERFEIÇOANDO O PROGRAMA PRNTCHRS

Após digitar, compilar e rodar o programa PRNTCHRS, notaremos que ele pode sofrer alguns aperfeiçoamentos. Ele não apresenta a função de retrocesso, para tornar possíveis as correções, e não oferece uma forma fácil para se colocar um caractere numa posição específica na tela. Além disso, não há cursor, e portanto não existe como dizer exatamente onde um caractere será apresentado.

Se rodarmos o PRNTCHRS em uma TV ou monitor coloridos, encontraremos ainda outro problema: as letras que o programa produz não são totalmente brancas, apresentando cores borradas nos contornos. Infelizmente, este "efeito arco-íris" é difícil de ser evitado quando caracteres pequenos são apresentados na tela do Apple IIc/IIe durante o modo de alta resolução.

Como veremos adiante, este problema é causado pelas técnicas de geração de cores utilizadas nos computadores Apple IIc/IIe quando em modo de alta resolução. Por razões que se tornarão claras no desenrolar deste capítulo, pontos ativados em determinadas colunas da tela, às vezes, apresentam uma cor, enquanto pontos na próxima coluna apresentam outra cor.

O problema não afeta o Apple IIc/IIe quando em modo de texto, que utiliza técnicas diferentes para gerar apresentação no vídeo. Se desejarmos misturar texto e gráficos em alta resolução, entretanto, existem duas formas simples para evitar o efeito arco-íris:

usarmos um monitor monocromático ou melhorarmos a resolução, apresentando caracteres de maiores dimensões.

O Programa 14-4, intitulado HEADLINES, oferece soluções, pelo menos parciais, para os problemas acima expostos. O programa, escrito em um Merlin Pro, inclui uma função de retrocesso, para correção de erros, e os caracteres que o programa produz podem ser colocados facilmente em qualquer lugar na tela, devido às suas dimensões, dimensões essas que também solucionam o problema do arco-íris.

Programa 14-4

O PROGRAMA HEADLINES

```
 1 *
 2 * HEADLINES.S
 3 *
 4  ORG $6FFD
 5 *
 6  JMP START
 7 *
 8 KYBD EQU $C000
 9 STROBE EQU $C010
10 *
11 TEMPLO EQU $06
12 TEMPHI EQU TEMPLO+1
13 CBASLO EQU TEMPHI+1
14 CBASHI EQU CBASLO+1
15 TABPTR EQU $4A
16 *
17 FILVAL EQU $300
18 TABSIZ EQU FILVAL+1
19 QUOT1 EQU TABSIZ+1
20 QUOT2 EQU QUOT1+1
21 REMDR1 EQU QUOT2+1
22 REMDR2 EQU REMDR1+1
23 YLINE EQU REMDR2+1
24 PRODL EQU YLINE+2
25 PRODH EQU PRODL+1
26 MPRL EQU PRODH+1
27 MPRH EQU MPRL+1
28 MPDL EQU MPRH+1
29 MPDH EQU MPDL+1
30 TOTAL EQU MPDH+1
31 XPSN EQU TOTAL+2
32 YPSN EQU XPSN+1
33 COUNT EQU YPSN+1
34 OB EQU COUNT+1
35 NB1 EQU OB+1
36 NB2 EQU NB1+1
37 YCOUNT EQU NB2+1
```



```
38 *
39 KILOBYTE EQU 1024
40 SCRTOP EQU $2000
41 *
42 PTRH EQU $4000
43 PTRL EQU $40C0
44 *
45 CHRTAB EQU *
46 *
47  HEX 2024282C3034383C2024282C3034383C
48  HEX 2125292D3135393D2125292D31355393D
49  HEX 22262A2E32363A3E22262A2E32363A3E
50  HEX 23272B2F33373B3F23272B2F33373B3F
51  HEX 00000000000000000808080808080808
52  HEX 00000000000000000808080808080808
53  HEX 00000000000000000808080808080808
54  HEX 00000000000000000808080808080808
55  HEX 2828282828282828A8A8A8A8A8A8A8A8
56  HEX 2828282828282828A8A8A8A8A8A8A8A8
57  HEX 2828282828282828A8A8A8A8A8A8A8A8
58  HEX 2828282828282828A8A8A8A8A8A8A8A8
59  HEX 5050505050505050D0D0D0D0D0D0D0D0
60  HEX 5050505050505050D0D0D0D0D0D0D0D0
61  HEX 5050505050505050D0D0D0D0D0D0D0D0
62  HEX 5050505050505050D0D0D0D0D0D0D0D0
63  HEX D000000000000000001C1C1C1C001C1C
64  HEX 00363624360000000123F3F123F3F12
65  HEX 000C3F033F303F0C000027170F3C3A39
66  HEX 00060902042A112E000C0C080C000000
67  HEX 00381C0E0E0E1C38000E1C3838381C0E
68  HEX 00082A1C3E1C2A08000C0C3F3F0C0C00
69  HEX 0000000000001C180C00003E3E000000
70  HEX 0000000000001C1C006070381C0E0703
71  HEX 001E33333333331E003C363330303030
72  HEX 001E3F33380E3F3F001E3F303E303F1E
73  HEX 00383C36333F3030003F3F031F303F1E
74  HEX 001C06031F33331E003F3F30180C0C0C
75  HEX 001E3F331E333F1E001E3F333E303E1E
76  HEX 00001C1C00001C1C00001C1C00001C18
77  HEX 0C70381C0E1C387000003E3E003E3E00
78  HEX 00070E1C381C0E07001E3F33180C00
79  HEX 0C007F7F7F7F7F7F7F7F1E3F333F3F33
80  HEX 33001F3F331F333F1F001E3F3303333F
81  HEX 1E001F3F3333333F1F003F3F031F033F
82  HEX 3F003F3F031F1F0303001E3F033B333F
83  HEX 1E003333333F3F3333003F3F0C0C0C3F
84  HEX 3F00B0B0B0B0B3BF9E80333B1F0F1F3B
85  HEX 330003030303033F3F0063776B636363
86  HEX 63003333373F3B3333001E3F3333333F
```

```
 87  HEX 1E001F3F333F1F0303001E3F23232B13
 88  HEX 2E001F3F333F1F3B33001E3F031E303F
 89  HEX 1E003F3F0C0C0C0C0C00333333333333F
 90  HEX 1E00333333333331E0C0063636366B6B77
 91  HEX 630033331E0C1E333003333331E0C0C
 92  HEX 0C003F3F180C063F3F001F1F0303031F
 93  HEX 1F0003070E1C387060001F1F1818181F
 94  HEX 1F000C1E3F0000000008080808080
 95  HEX 80FFFF060C183000000000000001E303E
 96  HEX 333E0003031F3333331F0000001E3303
 97  HEX 331E0030303E3333333E0000001E331F
 98  HEX 031E001C36061F0606060000001E3333
 99  HEX 3E301E03031F333333330000C000C0C
100  HEX 0C0C00300030303033331E0303331B0F
101  HEX 1B33000E0C0C0C0C0C1E0000003F5B5B
102  HEX 5B5B0000001F333333330000001E3333
103  HEX 331E0000001F33331F030300003E3333
104  HEX 3E303000001F33030303000000 1E031E
105  HEX 301E0006061F0606361C000000333333
106  HEX 333E0000003333331E0C0000006D6D6D
107  HEX 6D7E000000331E0C1E33000000333333
108  HEX 3E301E00003F180C063F001C1E060706
109  HEX 1E1C000C0C0C0C0C0C0C0C0C0E1E183818
110  HEX 1E0E000000281400000000FFFFFFFF
111 *
112 * ESTABELECER TABELA DE COORDENADAS DE LINHAS
113 *
114 START LDY #0
115 YLOOP CPY #192
116  BCC CONT
117  JMP EXIT
118 *
119 * DIVIDIR Y POR 8
120 *
121 CONT TYA
122  LSR
123  LSR
124  LSR
125  STA QUOT1
126 *
127 * OBTER RESTO DA DIVISAO POR 8
128 *
129  TYA
130  AND #7
131  STA REMDR1
132 *
133 * DIVIDIR QUOT1 POR 8
134 *
135  LDA QUOT1
136  LSR
```



```
137  LSR
138  LSR
139  STA QUOT2
140 *
141 * OBTER RESTO
142 *
143  LDA QUOT1
144  AND #7
145  STA REMDR2
146 *
147 * CALCULAR BYTE INFERIOR DO ENDERECO Y
148 *
149  LDA #0
150  STA MPRH
151  STA MPDH
152  LDA QUOT2
153  STA MPRL
154  LDA #40
155  STA MPDL
156  JSR MULT16
157  LDA PRODL
158  STA TOTAL
159  LDA PRODH
160  STA TOTAL+1
161
162  LDA #0
163  STA MPRH
164  STA MPDH
165  LDA REMDR2
166  STA MPRL
167  LDA #128
168  STA MPDL
169  JSR MULT16
170  CLC
171  LDA PRODL
172  ADC TOTAL
173  STA TOTAL
174  LDA PRODH
175  ADC TOTAL+1
176  STA TOTAL+1
177 *
178  LDA #0
179  STA MPRH
180  LDA REMDR1
181  STA MPRL
182  LDA #<KILOBYTE
183  STA MPDL
184  LDA #>KILOBYTE
185  STA MPDH
186  JSR MULT16
```

```
187   CLC
188   LDA PRODL
189   ADC TOTAL
190   STA TOTAL
191   LDA PRODH
192   ADC TOTAL+1
193   STA TOTAL+1
194  *
195   CLC
196   LDA #<SCRTOP
197   ADC TOTAL
198   STA PTRL,Y
199   LDA #>SCRTOP
200   ADC TOTAL+1
201   STA PTRH,Y
202  *
203   INY
204   JMP YLOOP
205  *
206  EXIT EQU *
207  *
208  *  INICIALIZAR APRESENTACAO NA TELA
209  *
210  INIT STA $C050 ;DESATIVAR MODO DE TEXTO
211   STA $C052 ;DESATIVAR MODO MIXTO
212   STA $C057 ;ATIVAR MODO DE ALTA RESOLUCAO
213  *
214  *  LIMPAR TELA
215  *
216   LDA #0
217   STA FILVAL
218   LDA #$00
219   STA TABPTR
220   STA TABSIZ
221   LDA #$20
222   STA TABPTR+1
223   STA TABSIZ+1
224   JSR BLKFIL
225  *
226  *  APRESENTAR CARACTERE NA TELA
227  *
228   LDA #0
229   STA XVALUE
230   STA XPSN
231   STA YVALUE
232   STA YPSN
233  *
234  SHOWCHRS JSR PRINTIT
235   JSR RESET
236   JMP SHOWCHRS
```



```
237  *
238  PRINTIT LDA KYBD
239   CMP #$80
240   BCC PRINTIT ;NENHUMA TECLA PRESSIONADA; TENTAR NOVAMENTE
241   STA STROBE
242   AND#$7F ;LIMPAR BIT SUPERIOR
243   CMP#8 ;SETA ESQUERDA PRESSIONADA?
244   BEQ BACKUP
245   CMP#127 ;APAGAR TECLA
246   BNE JUMP
247  *
248  BACKUP JSR MOVEBACK
249   LDA#32 ;ESPACO
250   STA CHAR
251   LDA XPSN
252   STA XVALUE
253   LDA YPSN
254   STA YVALUE
255   JSR PRNTCHRS
256   JSR MOVEBACK
257   RTS
258  *
259  MOVEBACK LDX XPSN
260   DEX
261   DEX
262   BPL LEAP
263   LDA YPSN
264   SEC
265   SBC #16
266   STA YPSN
267   LDX #38
268  LEAP STX XPSN
269   RTS
270  *
271  JUMP CMP#32 ;TECLA CONTROL PRESSIONADA?
272   BCC PRINTIT
273   STA CHAR
274   LDX XPSN
275   STX XVALUE
276   LDY YPSN
277   STY YVALUE
278   JSR PRNTCHRS
279   RTS
280  *
281  *  'RESET' NAS COORDENADAS DE TELA
282  *
283  RESET LDX XPSN
284   INX
285   INX
286   CPX #40
```

```
287   BCC NEXT
288   CLC
289   LDA YPSN
290   ADC #16
291   CMP #185
292   BCC ITSOK
293   LDA #0
294 ITSOK STA YPSN
295   LDX #0
296 NEXT STX XPSN
297   RTS
298 *
299 *  ROTINA DE MULTIPLICACAO EM 16 BITS
300 *
301 MULT16 LDA #0
302   STA PRODL
303   STA PRODH
304   LDX #16
305 SHIFT ASL PRODL
306   ROL PRODH
307   ASL MPRL
308   ROL MPRH
309   BCC NOADD
310   CLC
311   LDA MPDL
312   ADC PRODL
313   STA PRODL
314   LDA MPDH
315   ADC PRODH
316   STA PRODH
317 NOADD DEX
318   BNE SHIFT
319   RTS
320 *
321 *  ROTINA PRNTCHRS
322 *
323 PRNTCHRS LDA #<CHRTAB
324   STA CBASLO
325   LDA #>CHRTAB
326   STA CBASHI
327   LDA CHAR
328   LSR
329   LSR
330   LSR
331   LSR
332   LSR
333   CLC
334   ADC CBASHI
335   STA CBASHI
336   LDA CHAR
```



```
337   AND #$1F
338   ASL
339   ASL
340   ASL
341   CLC
342   ADC CBASLO
343   STA CBASLO
344   LDX #0
345   STX COUNT
346 LOOP LDY YVALUE
347   LDA PTRL,Y
348   STA TEMPLO
349   LDA PTRH,Y
350   STA TEMPHI
351   TXA
352   TAY
353   LDA (CBASLO),Y
354   STA OB
355 *
356 *  REPARTIR BYTE ORIGINAL EM NB1 E NB2
357 *
358   ASL OB
359 *
360   LDY #3
361 LOOPA ASL OB
362   ROL NB2
363   SEC
364   ROL NB2
365   DEY
366   BNE LOOPA
367   ASL OB
368   ROL NB2
369 *
370   LDY #3
371 LOOPB SEC
372   ROL NB1
373   ASL OB
374   ROL NB1
375   DEY
376   BNE LOOPB
377   SEC
378   ROL NB1
379 *
380   LDA NB1
381   ORA #$80
382   STA NB1
383   LDA NB2
384   ORA #$80
385   STA NB2
386 *
```

```
387 * DE VOLTA AO PROGRAMA ORIGINAL
388 *
389  LDA #2
390  STA YCOUNT
391 *
392 TWICE LDA NB1
393  LDY XVALUE
394  STA (TEMPLO),Y
395  INC XVALUE
396  LDA NB2
397  LDY XVALUE
398  STA (TEMPLO),Y
399  LDX XPSN
400  STX XVALUE
401  INC YVALUE
402 *
403  LDX YCOUNT
404  DEX
405  STX YCOUNT
406  BEQ HOP
407 *
408  LDX YVALUE
409  LDA PTRL,X
410  STA TEMPLO
411  LDA PTRH,X
412  STA TEMPHI
413  JMP TWICE
414 *
415 HOP LDX COUNT
416  INX
417  STX COUNT
418  CPX #8
419  BEQ SKIP
420  JMP LOOP
421 SKIP RTS
422 *
423 * ROTINA DE PREENCHIMENTO DE BLOCO
424 *
425 BLKFIL LDA FILVAL
426  LDX TABSIZ+1
427  BEQ PARTPG
428  LDY #0
429 FULLPG STA (TABPTR),Y
430  INY
431  BNE FULLPG
432  INC TABPTR+1
433  DEX
434  BNE FULLPG
435 PARTPG LDX TABSIZ
436  BEQ FINI
```



```
437  LDY #0
438 PARTLP STA (TABPTR),Y
439  INY
440  DEX
441  BNE PARTLP
442 FINI RTS
443 *
444 CHAR DFB 0
445 XVALUE DFB 0
446 YVALUE DFB 0
447 *
```

Para entender como funciona o programa HEADLINES, é preciso conhecer algo a respeito da forma pela qual o Apple IIc e o Apple IIe geram cores na tela. A Figura 14-3 mostra como opera o sistema de geração de cores.

Como se pode ver na figura, cada caractere apresentado em uma tela de alta resolução é representado por oito bytes de dados no mapa de bits. Entretanto, somente sete bits em cada byte são realmente apresentados na tela. O bit de ordem superior não é apresentado, e sim usado como um bit de cor. Em cada byte do mapa de tela, o bit de cor determina que cores serão usadas quando os outros bits do byte em questão forem apresentados na tela.

Com o auxílio do bit de cor em cada byte, seis cores podem ser apresentadas na tela de alta resolução: preto, branco, verde, violeta, laranja e azul.

Se os bits de um byte não precisam ser apresentados em cores, mas apenas em branco e preto, não fará diferença o fato do bit de cor estar ativado ou não. Se dois pontos contíguos forem acionados – isto é, se os dois bits adjacentes que os representam forem ativados para 1 – ambos os pontos serão apresentados em branco. Entretanto, se dois pontos contíguos forem desligados – desativando-se os bits que os representam – os pontos serão apresentados em preto.

Quando outras cores, que não branco e preto, tiverem de ser apresentadas, os bits não serão ativados ou desativados aos pares. Em vez disso, somente um bit é ativado para cada dois bits a serem apresentados na tela. Se um bit de numeração par é ativado e o bit à sua direita é desativado, ambos serão apresentados na cor indicada pelo bit ativado. Se um bit de numeração ímpar é desativado, e o bit à sua direita é ativado, ambos serão apresentados em cor, mas as cores serão diferentes.

Como já vimos, o bit de cor em um byte é também um fator determinante da cor. Se o bit de cor de um byte é ativado, as cores geradas pelos demais bits daquele byte serão laranja e azul. Se o bit de cor é desativado, as cores geradas serão verde e violeta.

Outro fator que é usado para determinar cores na tela é a posição da coluna de um dado byte na tela. Se o byte se encontra em uma coluna de numeração par, os bits pares produzirão uma cor e os ímpares, outra; se o byte se encontra em uma coluna ímpar, essas cores serão invertidas. Isto pode parecer uma forma estranha de disposição de um mapa de tela, mas faz sentido. Como somente 7 bits em cada byte são apresentados na tela, cada coluna da tela contém um número ímpar de pontos. Se o mesmo padrão ligado/desligado fosse utilizado para gerar as cores das colunas pares e ímpares, teríamos um par de bits ativados ou um par de bits desativados a cada encontro de colunas, e o resultado seria um par de pontos brancos ou um par de pontos pretos.

	Bytes em Colunas Pares								Bytes em Colunas Ímpares							
	0	1	2	3	4	5	6	7*	0	1	2	3	4	5	6	7*
Preto	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Preto	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
Branco	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0
Branco	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Verde	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0
Violeta	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	0
Laranja	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	1
Azul	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1

* Bit de Cor

Figura 14-3 Como o Apple IIc e IIe geram cores na tela.

Este problema foi evitado através da utilização de padrões alternados para a produção da mesma cor nas colunas pares e ímpares da tela, ou seja, um determinado padrão de pontos pode ser usado para gerar uma única cor através de toda a tela.

COMO OS DADOS DA TELA SÃO ARMAZENADOS EM RAM

Antes de passarmos a uma análise linha por linha do programa HEADLINES, há mais um ponto a ser observado. Por estranho que pareça, os pontos são reproduzidos na tela do Apple IIc/IIe *na ordem inversa em que são armazenados na RAM.*

Quando um byte é armazenado na RAM, o bit 0 (o bit de ordem inferior) está à direita, e o bit 7 (superior) à esquerda. Na tela, entretanto, o ponto representado pelo bit 0 está à esquerda, e o representado pelo bit 7, à direita. (O bit 7 – bit de cor – não é apresentado na tela.)

A Figura 14-4 mostra como um padrão de pontos para a letra B aparece na tela, como ele se apresenta quando convertido em dados de tela ligado/desligado, e como os 8 bytes usados para o caractere são armazenados na memória. Como se pode ver, os bytes armazenados em RAM são a imagem especular do padrão de bits correspondente à tela.

COMO FUNCIONA O PROGRAMA HEADLINES

Este programa é bastante similar ao programa PRNTCHRS. Na realidade, é o mesmo programa, com a adição de alguns aperfeiçoamentos.

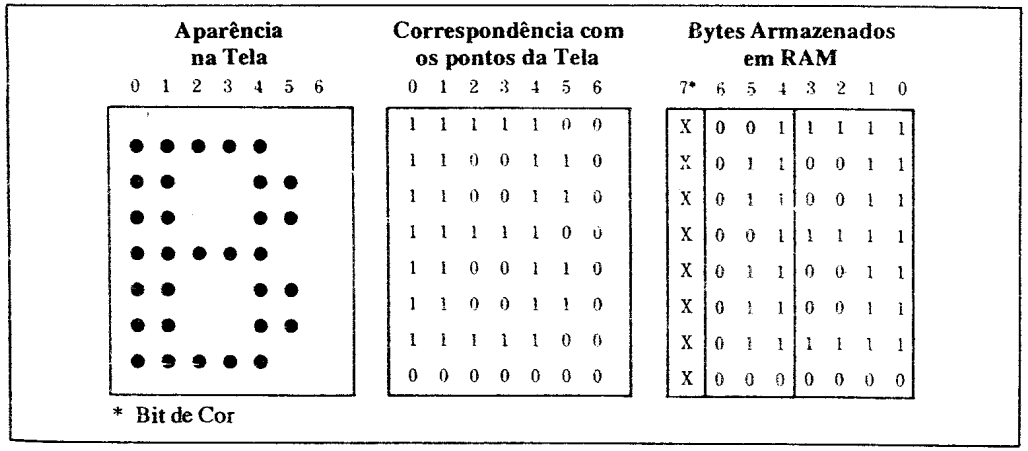


Figura 14-4 Como um caractere da tela é armazenado em RAM.

A diferença mais óbvia entre os dois programas encontra-se no fato dos caracteres apresentados pelo programa HEADLINES serem quatro vezes maiores, apesar de terem sido gerados pelo mesmo jogo de caracteres usado para o programa PRNTCHRS.

Outra diferença é que os caracteres, no HEADLINES, são apresentados contra um fundo azul, em lugar de um fundo preto. Além disso, os caracteres são totalmente brancos, com contornos nítidos, e não vagamente definidos e com arco-íris, como se mostravam no programa PRNTCHRS.

Mais uma diferença: o HEADLINES dispõe da função de retrocesso, permitindo correção de erros. E, embora o programa não apresente um cursor, não é difícil saber onde o próximo caractere aparecerá, uma vez que o fundo azul vai sendo criado à medida que vão sendo digitados os caracteres.

O sistema usado para aumentar o tamanho dos caracteres no programa HEADLINES não é difícil de entender. Nas linhas de 356 até 421, cada byte a ser apresentado na tela é expandido em dois bytes. O endereço de memória no qual está armazenado o byte original é intitulado OB, e os dois novos bytes usados para armazenar o byte expandido são chamados de NB1 e NB2.

Nas linhas de 356 a 421, OB é expandido em NB1 e NB2 através de uma série de instruções ASL e ROL. Conforme estas operações vão-se realizando, um bit ativado é inserido após cada bit do byte original. Esta inserção mantém os caracteres brancos, mas muda a cor do fundo para azul, já que o programa também ativa o bit de cor de cada byte, ativa os bits pares dos bytes pares e os bits ímpares dos bytes ímpares.

Expandido o byte original em NB1 e NB2, tornar o caractere duas vezes mais largo é fácil: os dois novos bytes são apresentados um após o outro. Após isso, a posição Y de NB1 e NB2 é incrementada para o endereço da próxima linha Y na tela e eles são apresentados novamente, obtendo-se um novo par de bytes abaixo do par que já se encontrava na tela. Desta forma, o caractere será também duas vezes mais alto do que no jogo de caracteres.

Quando um caractere acaba de ser apresentado neste sistema, uma rotina chamada RESET (linhas de 283 a 297) é usada para estabelecer as variáveis de tela XPST e YPST de maneira apropriada à apresentação de outro caractere.

Se as teclas DELETE ou SETA ESQUERDA forem pressionadas, porém, uma rotina chamada BACKUP (linhas de 248 a 269) é usada para retroceder um cursor invisível, apagar o último caractere digitado, colocando em seu lugar um espaço e retroceder novamente, para que ali possamos digitar um novo caractere.

Embora o HEADLINES tenha sido projetado para ser um programa de demonstração, pode ser facilmente expandido e transformado num interessante utilitário, incluindo-se uma rotina simples que permite o armazenamento, em disco, das telas geradas pelo programa, o que permitirá usá-las como telas-título ou outras aplicações.

GRÁFICOS DE DUPLA ALTA RESOLUÇÃO

Com o advento do Apple IIc e do Apple IIe tornaram-se acessíveis, aos programadores Apple, os gráficos de dupla alta resolução, que permitem a apresentação de até 16 cores em telas de alta resolução, ou o incremento da resolução horizontal monocromática para 560 pontos.

Além disso, a cor de cada ponto, em dupla alta resolução, pode ser controlada individualmente, não havendo restrições quanto às cores que podem ser "encostadas" a outra cor dentro de um mesmo byte – e a única restrição, quanto ao número de cores dentro de um byte, é o número de pontos disponíveis.

O Apple IIc/IIe usa uma técnica para geração de gráficos de dupla alta resolução que se assemelha à técnica utilizada para a apresentação de texto em 80 colunas. Quando se utiliza a dupla alta resolução, o Apple IIc/IIe apanha os dados do mesmo mapa de tela usado para os gráficos de alta resolução: Página Gráfica de Alta Resolução, 1, que se estende de $2000 a $3FFF.

No modo de dupla alta resolução, porém, os dados retirados da Página 1 podem vir tanto da memória principal quanto da auxiliar. Na tela de dupla alta resolução, os bytes da memória principal e da auxiliar são intercalados, da mesma forma que numa apresentação em 80 colunas. Uma tela de dupla alta resolução, como uma tela de texto em 80 colunas, tem 80 bytes de largura. Colunas pares de bytes, começando da Coluna $00, vêm da memória auxiliar. Colunas ímpares, partindo da $01, vêm da memória principal.

COMO SÃO PROGRAMADAS AS CORES EM DUPLA ALTA RESOLUÇÃO

O modo gráfico de dupla alta resolução gera cores de forma diferente da utilizada pelos gráficos de alta resolução normal. Embora também sejam apresentados na tela somente 7 bits de cada byte, em dupla alta resolução o bit superior de cada byte não é um bit de cor. E também não é apresentado na tela; simplesmente não é usado.

Em lugar de utilizar um bit de cor para estabelecer a cor de cada byte na tela, o modo de dupla alta resolução, apesar de dispor de uma resolução horizontal efetiva de 140 pontos – exatamente a mesma do modo de alta resolução – tem à disposição muito mais memória. Assim, em um gráfico de dupla alta resolução, 4 bits de memória (em lugar de um único bit) são usados para determinar a cor de cada ponto na tela. A Tabela 14-1 apresenta as cores usadas em gráficos de dupla alta resolução, com seus respectivos números-código, expressos em notação hexadecimal e binária.

Tabela 14-1 Cores Usadas em Gráficos de Dupla Alta Resolução

Cor	Código Hexa	Código Binário
Preto	$0	0000
Vermelho-escuro	$1	0001
Azul-escuro	$2	0010
Violeta	$3	0011
Verde-escuro	$4	0100
Cinza 1	$5	0101
Azul médio	$6	0110
Azul-claro	$7	0111
Marrom	$8	1000
Laranja	$9	1001
Cinza 2	$A	1010
Rosa	$B	1011
Verde	$C	1100
Amarelo	$D	1101
Verde-claro	$E	1110
Branco	$F	1111

Da mesma forma que em gráficos de alta resolução normal, os bits usados para os padrões de pontos são apresentados na ordem inversa à do armazenamento em RAM. Assim, os padrões de pontos usados para representar *cores* são também imagens especulares de seus padrões de bits reais. A Tabela 14-2 lista os códigos binários de cada cor e os padrões de pontos usados para apresentar cada cor na tela.

A Figura 14-5 ilustra como estes dados de cores são apresentados e codificados nos 80 bytes de memória de tela usados para armazenar cada linha de pontos da tela em RAM. Como são usados somente 7 bits de cada byte da memória de tela, os bits usados para gerar códigos de cores não se alinham aos bytes da forma como são armazenados. Este fato dificulta, até certo ponto, a escrita dos programas em dupla alta resolução.

Antes que um programa possa apresentar um determinado ponto, em uma cor desejada, precisa realizar uma série de cálculos razoavelmente complexos. Primeiramente,

deve calcular o número da coluna na qual o byte que contém o ponto aparecerá. Após isto, tem de determinar se os dados usados para apresentar aquela coluna virão da memória principal ou da auxiliar. O programa precisa também calcular a posição do bit desejado dentro do byte desejado. E esta é a parte mais difícil. Como os padrões de pontos usados para gerar cores não são alinhados com os limites dos bytes apresentados na tela, a disposição dos pontos necessários para desenhar uma cor diferem de byte para byte.

Especificamente, existem quatro pontos diferentes que podem ser usados para expressar cada uma das 16 cores, e o padrão de pontos a ser usado depende do número da coluna do byte no qual o ponto aparecerá.

Tabela 14-2 Padrões de Pontos para Cores em Gráficos de Dupla Alta Resolução

Cor	Código Binário	Padrão de Pontos
Preto	0000	0000
Vermelho-escuro	0001	1000
Azul-escuro	0010	0100
Violeta	0011	1100
Verde-escuro	0100	0010
Cinza 1	0101	1010
Azul médio	0110	0110
Azul-claro	0111	1110
Marrom	1000	0001
Laranja	1001	1001
Cinza 2	1010	0101
Rosa	1011	1101
Verde	1100	0011
Amarelo	1101	1011
Verde-claro	1110	0111
Branco	1111	1111

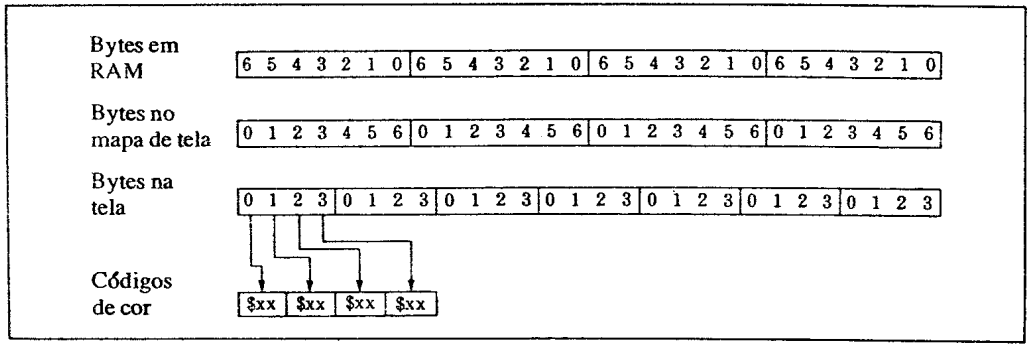


Figura 14-5 Como os bytes em RAM e códigos de cores são apresentados em gráficos de dupla alta resolução.

Uma forma de simplificar a escrita de programas gráficos em dupla alta resolução é a criação de uma tabela que contenha todos os números-código de cores utilizados em dupla alta resolução. Essa tabela poderá, então, ser inserida num programa em linguagem de máquina e consultada para obtenção do código adequado, cada vez que uma cor é utilizada. A Tabela 14-3 lista as 16 cores, com os quatro números-código de cada cor.

Tabela 14-3 Cores em Dupla Alta Resolução e respectivos Números-código

Cor	Memória Auxiliar, Colunas Pares	Memória Principal, Colunas Pares	Memória Auxiliar, Colunas Ímpares	Memória Principal Colunas Ímpares
Preto	$00	$00	$00	$00
Magenta	$08	$11	$22	$44
Marrom	$44	$08	$11	$22
Laranja	$4C	$19	$33	$66
Verde-escuro	$22	$44	$08	$11
Cinza 1	$2A	$55	$2A	$55
Verde	$66	$4C	$19	$33
Amarelo	$6E	$5D	$3B	$77
Azul-escuro	$11	$22	$44	$08
Roxo	$19	$33	$66	$4C
Cinza 2	$55	$2A	$55	$2A
Rosa	$5D	$3B	$77	$6E
Azul médio	$33	$66	$4C	$19
Azul-claro	$3B	$77	$6E	$5D
Turquesa	$77	$6E	$5D	$3B
Branco	$7F	$7F	$7F	$7F

Escrevendo um Programa em Dupla Alta Resolução O Programa 14-5, DUBHIRES, é um programa do tipo "digite e rode" que demonstra como as cores são programadas em dupla alta resolução. Foi escrito com um Merlin Pro – mas, como os demais programas deste capítulo, pode ser digitado e rodado em um Apple ProDOS Assembler, com modificações relativamente simples.

Programa 14-5
UM PROGRAMA GRÁFICO EM DUPLA ALTA RESOLUÇÃO

```
 1 *
 2 * DUBHIRES.S
 3 *
 4 * ESTE PROGRAMA APRESENTA AS 16 CORES QUE SAO
 5 * DISPONIVEIS EM GRAFICOS APPLE IIC/IIE DE DUPLA ALTA
     RESOLUCAO
 6 *
 7  ORG $8000
```



```
 8 *
 9  JMP INIT
10 *
11 KILOBYTE EQU 1024
12 SCRTOP EQU $2000
13 *
14 LOOKHI EQU $9000
15 LOOKLO EQU $90C0
16 *
17 * SOFT SWITCHES
18 *
19 COL80 EQU $C00D
20 TEXTOFF EQU $C050
21 MIXEDOFF EQU $C052
22 HIRES EQU $C057
23 STORE80 EQU $C001
24 PAGE2ON EQU $C055
25 PAGE2OFF EQU $C054
26 DHIRES EQU $C05E
27 *
28 * CONSTANTES DEFINIDAS PELO USUARIO
29 *
30 SCRPTR EQU $06
31 TEMPLO EQU SCRPTR+2
32 TEMPHI EQU TEMPLO+1
33 TABPTR EQU $1A
34 COLUMN EQU TABPTR+1
35 ROW EQU COLUMN+2
36 *
37 CLRFLG EQU $300
38 CLRPTR EQU CLRFLG+1
39 FILVAL EQU CLRPTR+1
40 TABSIZ EQU FILVAL+1
41 QUOT1 EQU TABSIZ+1
42 QUOT2 EQU QUOT1+1
43 REMDR1 EQU QUOT2+1
44 REMDR2 EQU REMDR1+1
45 YLINE EQU REMDR2+1
46 PRODL EQU YLINE+2
47 PRODH EQU PRODL+1
48 MPRL EQU PRODH+1
49 MPRH EQU MPRL+1
50 MPDL EQU MPRH+1
51 MPDH EQU MPDL+1
52 TOTAL EQU MPDH+1
53 XPSN EQU TOTAL+2
54 YPSN EQU XPSN+1
55 CODENR EQU YPSN+1
56 BARCNT EQU CODENR+1
57 *
```

```
58 COLORS EQU *
59 *
60 WHITE HEX 7F,7F,7F,7F
61 BLACK HEX 00,00,00,00
62 MAG HEX 08,11,22,44
63 BROWN HEX 44,08,11,22
64 ORANGE HEX 4C,19,33,66
65 DGREEN HEX 22,44,08,11
66 GRAY1 HEX 2A,55,2A,55
67 GREEN HEX 66,4C,19,33
68 YELLOW HEX 6E,5D,3B,77
69 DBLUE HEX 11,22,44,08
70 PURPLE HEX 19,33,66,4C
71 GRAY2 HEX 55,2A,55,2A
72 PINK HEX 5D,3B,77,6E
73 MBLUE HEX 33,66,4C,19
74 LBLUE HEX 3B,77,6E,5D
75 AQUA HEX 77,6E,5D,3B
76 *
77 * ESTABELECER DUPLA ALTA RESOLUCAO
78 *
79 INIT LDA #0
80  STA STORE80
81  STA TEXTOFF
82  STA MIXEDOFF
83  STA PAGE2OFF
84  STA HIRES
85  STA COL80
86  STA DHIRES
87 *
88 * INICIALIZAR PONTEIROS
89 *
90  LDA #0
91  STA COLUMN
92  STA ROW
93  STA BARCNT
94 *
95  LDA #<SCRTOP
96  STA SCRPTR
97  LDA #>SCRTOP
98  STA SCRPTR+1
99 *
100 * ESTABELECER TABELA DE ENDERECOS Y
101 *
102  JSR MAKETAB
103 *
104 * INICIO DO PROGRAMA PRINCIPAL
105 *
106  STA PAGE2ON
107  JSR PRNTROW
```



```
108  STA PAGE2OFF
109  JSR PRNTROW
110  RTS
111 *
112 PRNTROW LDY ROW
113  LDA LOOKLO,Y
114  STA TEMPLO
115  LDA LOOKHI,Y
116  STA TEMPHI
117 CLEARX LDX #0
118  LDY COLUMN
119 NXTCOL STA PAGE2ON
120  JSR PRINT
121  STA PAGE2OFF
122  JSR PRINT
123  INY
124  CPY#40;80 COLUNAS?
125  BCS ROWDUN
126  STY COLUMN
127  CPX #4
128  BCS CLEARX
129  JMP NXTCOL
130 ROWDUN LDX ROW
131  INX
132  CPX #192
133  BCS ALLDUN
134  JSR SETBAR
135  STX ROW
136  LDA #0
137  STA COLUMN
138  JMP PRNTROW
139 ALLDUN RTS
140 *
141 * ESTABELECER TABELA DE COORDENADAS DE LINHAS
142 *
143 MAKETAB LDY #0
144 YLOOP CPY #192
145  BCC MOVEON
146  JMP EXIT
147 *
148 * DIVIDIR Y POR 8
149 *
150 MOVEON TYA
151  LSR
152  LSR
153  LSR
154  STA QUOT1
155 *
156 * OBTER RESTO DA DIVISAO POR 8
157 *
```

```
158   TYA
159   AND #7
160   STA REMDR1
161  *
162  * DIVIDIR QUOT1 POR 8
163  *
164   LDA QUOT1
165   LSR
166   LSR
167   LSR
168   STA QUOT2
169  *
170  * OBTER RESTO
171  *
172   LDA QUOT1
173   AND #7
174   STA REMDR2
175  *
176  * CALCULAR BYTE INFERIOR DO ENDERECO Y
177  *
178   LDA #0
179   STA MPRH
180   STA MPDH
181   LDA QUOT2
182   STA MPRL
183   LDA #40
184   STA MPDL
185   JSR MULT16
186   LDA PRODL
187   STA TOTAL
188   LDA PRODH
189   STA TOTAL+1
190  *
191   LDA #0
192   STA MPRH
193   STA MPDH
194   LDA REMDR2
195   STA MPRL
196   LDA #128
197   STA MPDL
198   JSR MULT16
199   CLC
200   LDA PRODL
201   ADC TOTAL
202   STA TOTAL
203   LDA PRODH
204   ADC TOTAL+1
205   STA TOTAL+1
206  *
207   LDA #0
```



```
208   STA MPRH
209   LDA REMDR1
210   STA MPRL
211   LDA #<KILOBYTE
212   STA MPDL
213   LDA #>KILOBYTE
214   STA MPDH
215   JSR MULT16
216   CLC
217   LDA PRODL
218   ADC TOTAL
219   STA TOTAL
220   LDA PRODH
221   ADC TOTAL+1
222   STA TOTAL+1
223  *
224   CLC
225   LDA #<SCRTOP
226   ADC TOTAL
227   STA LOOKLO,Y
228   LDA #>SCRTOP
229   ADC TOTAL+1
230   STA LOOKHI,Y
231  *
232   INY
233   JMP YLOOP
234  *
235  EXIT RTS
236  *
237  *  ROTINA DE MULTIPLICACAO EM 16 BITS
238  *
239  MULT16 LDA #0
240   STA PRODL
241   STA PRODH
242   LDX #16
243  SHIFT ASL PRODL
244   ROL PRODH
245   ASL MPRL
246   ROL MPRH
247   BCC NOADD
248   CLC
249   LDA MPDL
250   ADC PRODL
251   STA PRODL
252   LDA MPDH
253   ADC PRODH
254   STA PRODH
255  NOADD DEX
256   BNE SHIFT
257   RTS
```

```
258 *
259 *  ROTINA AUTOMODIFICAVEL DE APRESENTACAO DE CORES
260 *
261 PRINT LDA COLORS,X
262   STA (TEMPLO),Y
263   INX
264   RTS
265 *
266 *  ROTINA PARA ESTABELECER NOVA BARRA DE COR
267 *
268 SETBAR TXA
269   PHA
270   LDX BARCNT
271   INX
272   CPX #12
273   BCC LEAP
274   LDA PRINT+1
275   CLC
276   ADC #4
277   STA PRINT+1
278   LDA PRINT+2
279   ADC #0
280   STA PRINT+2
281   LDX #0
282 LEAP STX BARCNT
283   PLA
284   TAX
285   RTS
```

O programa DUBHIRES começa na linha 9, com uma instrução que salta sobre o bloco introdutório de dados e conduz o programa à linha 79, intitulada INIT. Este bloco de instruções, primeiro segmento de código executável do programa, ativa um grupo de *soft switches* que inicializam o modo de dupla alta resolução. Todos eles residem na Página $C0 da memória. O *switch* chamado DHIRES, situado no endereço $C05E, controla um anunciador que permite o uso de gráficos de dupla alta resolução em lugar daquilo que seria, ordinariamente, o modo de texto em 80 colunas. (O uso dos demais *soft switches* utilizados nas linhas de 79 a 86 foi explicado no Capítulo 12.)

O *switch* TEXTOFF desativa o modo de texto do computador e completa o trabalho de possibilitar a utilização de gráficos. O *switch* MIXEDOFF desativa a janela de texto da parte inferior da tela, o COL80 ativa o firmware de 80 colunas, e o STORE80 determina se o *switch* PAGE2OFF será usado para efetuar o chaveamento entre os Displays de Texto e Alta Resolução, Páginas 1 e 2, na memória principal e auxiliar.

Como o modo gráfico de dupla alta resolução intercala as páginas da memória

principal e auxiliar, o STORE80, no programa DUBHIRES, é quem efetua o chaveamento entre as memórias.

Nas linhas de 88 a 99 do programa, alguns ponteiros importantes são inicializados. A seguir, na linha 102, uma sub-rotina que estabelece uma tabela de endereços Y é chamada. Esta sub-rotina, intitulada MAKETAB, trabalha exatamente como as rotinas que foram usadas nos programas PRNTCHRS e HEADLINES. No programa DUBHIRES, entretanto, a sub-rotina é utilizada para procurar endereços em dois segmentos de memória – a página de alta resolução na memória principal e a página de alta resolução na memória auxiliar.

A parte principal do programa constitui-se de apenas 5 linhas: da 106 a 110. O programa faz uso de uma longa rotina de elaboração de tabela, que inicia na linha 143, além de outras três sub-rotinas: a que gera linhas individuais de cores, a que escolhe o número de código para cada cor e a que divide as cores apresentadas na tela em 16 barras horizontais.

A sub-rotina que apresenta linhas individuais de cores denomina-se PRNTROW, e vai da linha 112 à 139. Este bloco de código consiste num *loop* que desenha uma linha colorida da esquerda para a direita, na tela, chamando a rotina MAKETAB para determinar o endereço de memória no qual cada byte daquela linha deve ser armazenado, de forma a aparecer na posição correta na tela.

As cores, no programa, são obtidas da tabela de cores que aparece entre as linhas 58 e 69. Os códigos de cores fornecidos por esta tabela são procurados e apresentados na tela por uma sub-rotina denominada PRINT, existente na linha 261. Essa sub-rotina é chamada pela linha 120 do programa.

A sub-rotina PRINT merece destaque, por utilizar um recurso de programação chamado *modificação de endereço*. Rotinas em assembly que usam essa técnica são, em geral, chamadas de rotinas *automodificáveis*.

Quando o programa DUBHIRES é carregado na memória, a sub-rotina PRINT usa endereçamento indexado para apresentar a cor branca na tela. Como se pode observar, esta cor provém de uma linha de dados denominada COLOR – especificamente, a seqüência de dados na linha 60 do programa.

Após as rotinas PRINTROW e PRINT apresentarem uma barra branca na tela, outra rotina, SETBAR, é chamada. Esta rotina, através de um *loop*, muda a cor que está sendo apresentada, de forma que uma barra de outra cor possa ser mostrada. Este é o ponto onde a modificação de endereço entra em cena.

Para compreender como ela funciona, convém considerar a relação entre a linguagem assembly e a linguagem de máquina. Como vimos, a linha 261 do DUBHIRES é intitulada PRINT. Assim, quando o programa é convertido em linguagem de máquina, o primeiro byte de código de máquina na linha intitulada PRINT será uma instrução em linguagem de máquina que equivale ao mnemônico LDA em assembly.

Vejamos, agora, o comando LDA COLORS,X da linha 261. Este comando foi escrito no modo de endereçamento conhecido como *endereçamento absoluto indexado*. De acordo com as regras de endereçamento em assembly do Apple IIc/IIe, o modo absoluto indexado sempre usa um operando de dois bytes. Portanto, o operando COLORS,X é um operando de dois bytes.

Quando o programa DUBHIRES é convertido em linguagem de máquina, a instrução em linguagem assembly LDA (que significa "carregar o acumulador") será convertida no op code $BD em linguagem de máquina. Quando este código operacional é encontrado, num programa em linguagem de máquina, o valor do registro X é somado ao endereço especificado, pelo operando de dois bytes que segue o op code, e o resultado deste cálculo torna-se o endereço efetivo da instrução LDA. Determinado o endereço efetivo, o valor armazenado nesse endereço é carregado no acumulador.

Como o operando COLORS,X segue uma instrução chamada PRINT, o primeiro byte do operando COLORS,X pode também ser designado PRINT+1. Da mesma forma, o segundo byte do operando COLORS,X pode sofrer a designação PRINT+2. E é exatamente assim que o programa se refere a esses bytes, nas linhas 274 a 280, a secção do programa que faz uso da técnica de modificação de endereços.

Compreendida esta técnica, não é difícil entender como ela é usada no programa DUBHIRES. Na linha 274, o número 4 é somado ao conteúdo de dois bytes, denominados PRINT+1 e PRINT+2 (em outras palavras, ao operando da instrução LDA da linha 261). Assim, na próxima vez que a rotina PRINT for executada, o acumulador não será carregado com o valor original de COLORS,X, e sim com esse valor *mais* 4. Na próxima vez, o valor de COLORS,X será novamente acrescido de 4, e assim por diante.

Código automodificável, quando adequadamente utilizado, pode economizar tempo e memória num programa em assembly, e também servir como uma boa alternativa para o endereçamento indireto indexado, quando o registro Y está sendo usado para outras finalidades, como no caso da rotina PRINT do programa DUBHIRES.

TABELA DE CONVERSÃO DE LINGUAGEM ASSEMBLY PARA LINGUAGEM DE MÁQUINA

Mnemônico	Endereço	Formato	Função	Tradução
ADC	61	Ind,X	Add with carry	Somar com transporte
ADC		65	Zpg	
ADC	69	Imm		
ADC		6D	Abs	
ADC	71	Ind, Y		
ADC	72*	(Zpg)*		
ADC	75	Zpg,X		
ADC	79	Abs,Y		
ADC	7D	Abs,X		
AND	21	Ind,X	Logical AND	E Lógico
AND		25	Zpg	
AND	29	Imm		
AND	2D	Abs		
AND	31	Ind,Y		
AND	32*	(Zpg)*		
AND		35	Zpg,X	
AND	39	Abs,Y		
AND	3D	Abs,X		
ASL	06	Zpg	Arithmetic shift left	Deslocamento aritmético à esquerda
ASL	OA	Acc		
ASL	OE	Abs		

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

Mnemônico	Endereço	Formato	Função	
ASL	16	Zpg,X		
ASL	1E	Abs,X		
BCC	90	Rel	Branch if carry clear	Desviar se o transporte estiver desativado
BCS	B0	Rel	Branch if carry set	Desviar se o transporte estiver ativado
BEQ	F0	Rel	Branch if equal to zero	Desviar se igual a zero
BIT	24	Zpg	Compare memory bits with accumulator	Comparar bits da memória com o acumulador
BIT	2C	Abs		
BIT	34*	Zpg,X		
BIT	3C*	Abs,X		
BIT	89*	Imm		
BMI	30	Rel	Branch on minus	Desviar se negativo
BNE	D0	Rel	Branch if not equal to zero	Desviar se diferente de zero
BPL	10	Rel	Branch on plus	Desviar se positivo
BRA*	80*	Rel	Branch always	Desviar sempre
BRK	00	Imp	Force break	Parada forçada
BVC	50	Rel	Branch if overflow clear	Desviar se o "overflow" estiver desativado
BVS	70	Rel	Branch if overflow set	Desviar se o "overflow" estiver ativado
CLC	18	Imp	Clear carry flag	Desativar indicador de transporte
CLD	D8	Imp	Clear decimal flag	Desativar indicador de decimal
CLI	58	Imp	Clear interrupt flag	Desativar indicador de interrupção
CLV	B8	Imp	Clear overflow flag	Desativar indicador de "overflow"
CMP	C1	Ind,X	Compare memory with accumulator	Comparar memória com o acumulador
CMP	C5	Zpg		
CMP	C9	Imm		
CMP	CD	Abs		
CMP	D1	Ind,Y		
CMP	D2*	(Zpg)*		
CMP	D5	Zpg,X		
CMP	D9	Abs,Y		

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

Mnemônico	Endereço	Formato	Função	
CMP	DD	Abs,X		
CPX	E0	Imm	Compare memory with X register	Comparar memória com o registro X
CPX	E4	Zpg		
CPX	EC	Abs		
CPY	C0	Imm	Compare memory with Y register	Comparar memória com o registro Y
CPY	C4	Zpg		
CPY	CC	Abs		
DEA ou				
DEC A*	3A*	Acc	Decrement accumulator	Decrementar o acumulador
DEC	C6	Zpg	Decrement memory	Decrementar memória
DEC	CE	Abs		
DEC	D6	Zpg,X		
DEC	DE	Abs,X		
DEX	CA	Imp	Decrement X register	Decrementar registro X
DEY	88	Imp	Decrement Y register	Decrementar registro Y
EOR	41	Ind,X	Exclusive EOR	OU Exclusivo
EOR	45	Zpg		
EOR	49	Imm		
EOR	4D	Abs		
EOR	51	Ind,Y		
EOR	52*	(Zpg)*		
EOR	55	Zpg,X		
EOR	59	Abs,Y		
EOR	5D	Abs,X		
INA ou				
INC A*	1A*	Acc	Increment accumulator	Incrementar acumulador
INC	E6	Zpg	Increment memory	Incrementar memória
INC	EE	Abs		
INC F5	F6	Zpg,X		

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

Mnemônico	Endereço	Formato	Função	
INC	FE	Abs,X		
INX	E8	Imp	Increment X register	Incrementar registro X
INY	C8	Imp	Increment Y register	Incrementar registro Y
JMP	4C	Abs	Jump to address	Pular para o endereço
JMP	6C	(Abs)		
JMP	7C*	Abs(Ind,X)*		
JSR	20	Abs	Jump to subroutine	Pular para a sub-rotina
LDA	A1	Ind,X	Load accumulator	Carregar o acumulador
LDA	A5	Zpg		
LDA	A9	Imm		
LDA	AD	Abs		
LDA	B1	Ind,Y		
LDA	B2*	(Zpg)*		
LDA	B5	Zpg,X		
LDA	B9	Abs,Y		
LDA	BD	Abs,X		
LDX	A2	Imm	Load X register	Carregar registro X
LDX	A6	Zpg		
LDX	AE	Abs		
LDX	BE	Abs,Y		
LDY	A0	Imm	Load Y register	Carregar registro Y
LDY	A4	Zpg		
LDY	AC	Abs		
LDY	B4	Zpg,X		
LSR	46	Zpg	Logical shift right	Deslocamento lógico à direita
LSR	4A	Acc		
LSR	4E	Abs		
LSR	56	Zpg,X		
LSR	5E	Abs,X		

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

Mnemônico	Endereço	Formato	Função	
NOP	EA	Imp	No operation	Nenhuma operação
ORA	01	Ind,X	Logical OR	OU Lógico
ORA	05	Zpg		
ORA	09	Imm		
ORA	0D	Abs		
ORA	11	Ind,Y		
ORA	12*	(Zpg)*		
ORA	15	Zpg,X		
ORA	19	Abs,Y		
ORA	1D	Abs,X		
PHA	48	Imp	Push accumulator	Empurrar o acumulador
PHP	08	Imp	Push processor (P) register	Empurrar o registro de status do processador (P)
PHX*	DA*	Imp	Push X register	Empurrar o registro X
PHY*	5A*	Imp	Push Y register	Empurrar o registro Y
PLA	68	Imp	Pull accumulator	Puxar o acumulador
PLP	28	Imp	Pull processor status (P) register	Puxar o registro de status do processador (P)
PLX*	FA*	Imp	Pull X register	Puxar registro X
PLY*	7A*	Imp	Pull Y register	Puxar registro Y
ROL	26	Zpg	Rotate left	Rotação à esquerda
ROL	2A	Acc		
ROL	2E	Abs		
ROL	3E	Abs,X		
ROL	36	ZX		
ROR	66	Zpg	Rotate right	Rotação à direita
ROR	6A	Acc		
ROR	6E	Abs		
ROR	76	Zpg,X		
ROR	7E	Abs,X		
RTI	40	Imp	Return from interrupt	Retornar da interrupção

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

Mnemônico	Endereço	Formato	Função	
RTS	60	Imp	Return from subroutine	Retornar da sub-rotina
SBC	E1	Ind,X	Subtract with carry	Subtrair com transporte
SBC	E5	Zpg		
SBC	E9	Imm		
SBC	ED	Abs		
SBC	F1	Ind, Y		
SBC	F2*	(Zpg)*		
SBC	F5	Zpg,X		
SBC	F9	Abs,X		
SBC	FD	Abs,Y		
SEC	38	Imp	Set carry flag	Ativar indicador de transporte
SED	F8	Imp	Set decimal flag	Ativar indicador de decimal
SEI	78	Imp	Set interrupt flag	Ativar indicador de interrupção
STA	81	Ind,X	Store accumulator	Armazenar acumulador
STA	85	Zpg		
STA	8D	Abs		
STA	91	Ind,Y		
STA	92*	(Zpg)*		
STA	95	Zpg,X		
STA	99	Abs,Y		
STA	9D	Abs,X		
STX	86	Zpg	Store X register	Armazenar registro X
STX	8E	Abs		
STX	96	Zpg,Y		
STY	84	Zpg		
STY	8C	Abs		
STY	94	Zpg,X		
STZ*	64*	Zpg	Store zero	Armazenar zero
STZ*	74*	Zpg,X		

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

Mnemônico	Endereço	Formato	Função	
STZ*	9C*	Abs		
STZ*	9E*	Abs,X		
TAX	AA	Imp	Transfer A to X	Transferir acumulador para X
TAY	A8	Imp	Transfer A to Y	Transferir acumulador para Y
TRB*	14*	Zpg	Test and reset bits	Verificar e desativar bits
TRB*	1C*	Abs		
TSB*	04*	Zpg	Test and set bits	Verificar e ativar bits
TSB*	0C*	Abs		
TSX	BA	Imp	Transfer stack pointer to X	Transferir ponteiro da pilha para X
TXA	8A	Imp	Transfer X to A	Transferir X para acumulador
TXS	9A	Imp	Transfer X to stack pointer	Transferir X para o ponteiro da pilha
TYA	98	Imp	Transfer Y to A	Transferir Y para acumulador

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

B

TABELA DE CONVERSÃO DE LINGUAGEM DE MÁQUINA PARA LINGUAGEM ASSEMBLY

Código-objeto	Mnemônico	Endereço
00	BRK	Imp
01	ORA	Ind,X
04*	TSB*	Zpg
05	ORA	Zpg
06	ASL	Zpg
08	PHP	Imp
09	ORA	Imm
0A	ASL	Acc
0C*	TSB*	Abs
0D	ORA	Abs
0E	ASL	Abs
10	BPL	Rel
11	ORA	Ind,Y
12*	ORA	(Zpg)*
14*	TRB*	Zpg
15	ORA	Zpg,X
16	ASL	Zpg,X
18	CLC	Imp
19	ORA	Abs,Y
1A*	INA ou INC A*	Acc
1C*	TRB*	Abs
1D	ORA	Abs,X
1E	ASL	Abs,X
20	JSR	Abs
21	AND	Ind,X

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

Código-objeto	Mnemônico	Endereço
24	BIT	Zpg
25	AND	Zpg
26	ROL	Zpg
28	PLP	Imp
29	AND	Imm
2A	ROL	Acc
2C	BIT	Abs
2D	AND	Abs
2E	ROL	Abs
30	BMI	Rel
31	AND	Ind,Y
32*	AND	(Zpg)*
34*	BIT	Zpg,X
35	AND	Zpg,X
36	ROL	ZX
38	SEC	Imp
39	AND	Abs,Y
3A*	DEA ou DEC A*	Acc
3C*	BIT	Abs,X
3D	AND	Abs,X
3E	ROL	Abs,X
40	RTI	Imp
41	EOR	Ind,X
45	EOR	Zpg
46	LSR	Zpg
48	PHA	Imp
49	EOR	Imm
4A	LSR	Acc
4C	JMP	Abs
4D	EOR	Abs
4E	LSR	Abs
50	BVC	Rel
51	EOR	Ind,Y
52*	EOR	(Zpg)*
55	EOR	Zpg,X
56	LSR	Zpg,X
58	CLI	Imp
59	EOR	Abs,Y
5A*	PHY*	Imp
5D	EOR	Abs,X
5E	LSR	Abs,X
60	RTS	Imp
61	ADC	Ind,X

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

Código-objeto	Mnemônico	Endereço
64*	STZ*	Zpg
65	ADC	Zpg
66	ROR	Zpg
68	PLA	Imp
69	ADC	Imm
6A	ROR	Acc
6C	JMP	(Abs)
6D	ADC	Abs
6E	ROR	Abs
70	BVS	Rel
71	ADC	Ind,Y
72*	ADC	(Zpg)*
74*	STZ*	Zpg,X
75	ADC	Zpg,X
76	ROR	Zpg,X
78	SEI	Imp
79	ADC	Abs,Y
7A*	PLY*	Imp
7C*	JMP	Abs(Ind,X)*
7D	ADC	Abs,X
/E	ROR	Abs,X
80	BRA*	Rel
81	STA	Ind,X
84	STY	Zpg
85	STA	Zpg
86	STX	Zpg
88	DEY	Imp
89*	BIT	Imm
8A	TXA	Imp
8C	STY	Abs
8D	STA	Abs
8E	STX	Abs
90	BCC	Rel
91	STA	Ind,Y
92*	STA	(Zpg)*
94	STY	Zpg,X
95	STA	Zpg,X
96	STX	Zpg,Y
98	TYA	Imp
99	STA	Abs,Y
9A	TXS	Imp
9C*	STZ*	Abs
9D	STA	Abs,X
9E*	STZ*	Abs,X

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

Código-objeto	Mnemônico	Endereço
A0	LDY	Imm
A1	LDA	Ind,X
A2	LDX	Imm
A4	LDY	Zpg
A5	LDA	Zpg
A6	LDX	Zpg
A8	TAY	Imp
A9	LDA	Imm
AA	TAX	Imp
AC	LDY	Abs
AD	LDA	Abs
AE	LDX	Abs
B0	BCS	Rel
B1	LDA	Ind,Y
B2*	LDA	(Zpg)*
B4	LDY	Zpg,X
B5	LDA	Zpg,X
B6	LDX	Zpg,Y
B8	CLV	Imp
B9	LDA	Abs,Y
BA	TSX	Imp
BC	LDY	Abs,X
BC	LDY	Abs,X
BD	LDA	Abs,X
BE	LDX	Abs,Y
C0	CPY	Imm
C1	CMP	Ind,X
C4	CPY	Zpg
C5	CMP	Zpg
C6	DEC	Zpg
C8	INY	Imp
C9	CMP	Imm
CA	DEX	Imp
CC	CPY	Abs
CD	CMP	Abs
CE	DEC	Abs
D0	BNE	Rel
D1	CMP	Ind,Y
D2*	CMP	(Zpg)*
D5	CMP	Zpg,X
D6	DEC	Zpg,X
D8	CLD	Imp
D9	CMP	Abs,Y

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

DA*	PHX*	Imp
DD	CMP	Abs,X
DE	DEC	Abs,X
E0	CPX	Imm
E1	SBC	Ind.X
E4	CPX	Zpg
E5	SBC	Zpg
E6	INC	Zpg
E8	INX	Imp
E9	SBC	Imm
EA	NOP	Imp
EC	CPX	Abs
ED	SBC	Abs
EE	INC	Abs
F0	BEQ	Rel
F1	SBC	Ind,Y
F2*	SBC	(Zpg)*
F5	SBC	Zpg,X
F6	INC	Zpg,X
F8	SED	Imp
F9	SBC	Abs,Y
FA*	PLX*	Imp
FD	SBC	Abs,X
FE	INC	Abs,X

* (Na primeira coluna) Novo mnemônico
(Na segunda coluna) Novo op code
(Na terceira coluna) Novo modo de endereçamento

C

INSTRUÇÕES DO 65C02

	ADC	Add Memory to Accumulator with Carry	Somar Memória ao Acumulador, com Transporte
	AND	"AND" Memory with Accumulator	"E" da Memória com o Acumulador
	ASL	Shift Left One Bit (Memory or Accumulator)	Deslocar um Bit à Esquerda (Memória ou Acumulador)
	BCC	Branch on Carry Clear	Desviar se o Transporte Estiver Desativado
	BCS	Branch on Carry Set	Desviar se o Transporte Estiver Ativado
	BEQ	Branch if Equal (Zero Flag Set)	Desviar se Igual (Indicador de Zero Ativado)
*	BIT	Test Bits in Memory with Accumulator	Verificar Bits da Memória com os do Acumulador
	BMI	Branch on Result Minus	Desviar se o Resultado for Negativo
	BNE	Branch on Result Not Zero	Desviar se o Resultado for Diferente de Zero
	BPL	Branch on Result Plus	Desviar se o Resultado for Positivo
•	BRA	Branch Relative Always	Desviar Sempre
	BRK	Force Break	Parada Forçada
	BVC	Branch on Overflow Clear	Desviar se o "Overflow" Estiver Desativado
	BVS	Branch on Overflow Set	Desviar se o "Overflow" Estiver Ativado
	CLC	Clear Carry Flag	Desativar Indicador de Transporte
	CLD	Clear Decimal Mode	Desativar Modo Decimal
	CLI	Clear Interrupt Disable Bit	Desativar Bit de Desativamento de Interrupções
	CLV	Clear Overflow Flag	Desativar Indicador de "Overflow"
*	CMP	Compare Memory and Accumulator	Comparar Memória com o Acumulador
	CPX	Compare Memory and Index X	Comparar Memória com o Índice X
	CPY	Compare Memory and Index Y	Comparar Memória com o Índice Y
*	DEC	Decrement Memory by One	Decrementar Memória em uma Unidade
	DEX	Decrement Index X by One	Decrementar o Índice X em uma Unidade
	DEY	Decrement Index Y by One	Decrementar o Índice Y em uma Unidade
*	EOR	"Exclusive-or" Memory with Accumulator	"OU Exclusivo" entre Memória e Acumulador
*	INC	Increment Memory by One	Incrementar a Memória em uma Unidade
	INX	Increment Index X by One	Incrementar o Índice X em uma Unidade
	INY	Increment Index Y by One	Incrementar o Índice Y em uma Unidade
*	JMP	Jump to New Location	Pular para uma Nova Posição

	JSR	Jump to New Location Saving Return Address / Pular para uma Nova Posição, Guardando o Endereço de Retorno
*	LDA	Load Accumulator with Memory / Carregar o Acumulador com a Memória
	LDX	Load Index X with Memory / Carregar o Índice X com a Memória
	LDY	Load Index Y with Memory / Carregar o Índice Y com a Memória
	LSR	Shift One Bit Right (Memory or Accumulator) / Deslocar um Bit à Direita (Memória ou Acumulador)
	NOP	No Operation / Nenhuma Operação
*	ORA	"OR" Memory with Accumulator / "OU" entre Memória e Acumulador
	PHA	Push Accumulator on Stack / Empurrar o Acumulador para a Pilha
	PHP	Push Processor Status on Stack / Empurrar o Registro de Status do Processador para a Pilha
•	PHX	Push Index X on Stack / Empurrar o índice X para a Pilha
•	PHY	Push Index Y on Stack / Empurrar o índice Y para a Pilha
	PLA	Pull Accumulator from Stack / Puxar da Pilha para o Acumulador
	PLP	Pull Processor Status from Stack / Puxar da Pilha para o Registro de Status do Processador
•	PLX	Pull Index X from Stack / Puxar o índice X da Pilha
•	PLY	Pull Index Y from Stack / Puxar o índice Y da Pilha
	ROL	Rotate One Bit Left (Memory or Accumulator) / Rotação de um Bit para a Esquerda (Memória ou Acumulador)
	ROR	Rotate One Bit Right / Rotação de um Bit para a Direita (Memória ou Acumulador)
	RTI	Return from Interrupt / Retornar da Interrupção
	RTS	Return from Subroutine / Retornar da Sub-rotina
*	SBC	Subtract Memory from Accumulator with Borrow / Subtrair Memória do Acumulador, com Empréstimo
	SEC	Set Carry Flag / Ativar Indicador de Transporte
	SED	Set Decimal Mode / Ativar Modo Decimal
	SEI	Set Interrupt Disable Status / Ativar Desabilitação de Interrupções
*	STA	Store Accumulator in Memory / Armazenar o Acumulador na Memória
	STX	Store Index X in Memory / Armazenar o Índice X na Memória
	STY	Store Index Y in Memory / Armazenar o Índice Y na Memória
•	STZ	Store Zero in Memory (Direct; Direct, X; Abs; Abs, X) / Armazenar Zero na Memória (Direto; Direto X; Abs; Abs, X)
	TAX	Transfer Accumulator to Index X / Transferir o Acumulador para o Índice X
	TAY	Transfer Accumulator to Index Y / Transferir o Acumulador para o Índice Y
•	TRB	Test and Reset Memory Bits Determined by the Accumulator A (Direct and Absolute) / Verificar e Desativar Bits da Memória, Determinados pelo Acumulador A (Direto e Absoluto)
•	TSB	Test and Set Memory Bits Bits Determined by the Accumulator A (Direct and Absolute) / Verificar e Ativar Bits da Memória, Determinados pelo Acumulador A (Direto e Absoluto)
	TSX	Transfer Stack Pointer to Index X / Transferir o Ponteiro da Pilha para o Índice X
	TXA	Transfer Index X to Accumulator / Transferir o Índice X para o Acumulador
	TXS	Transfer Index X to Stack Pointer / Transferir o Índice X para o Registro da Pilha
	TYA	Transfer Index Y to Accumulator / Transferir o Índice Y para o Acumulador

Nota: © Wester Design Center, Inc. usado sob permissão.

D

TABELA DE OP CODES DO 65C02

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK	ORA ind, X			TSB • zpg	ORA zpg	ASL zpg		PHP	ORA imm	ASL A		TSB • abs	ORA abs	ASL abs		0
1	BPL rel	ORA ind, Y	ORA • ind		TRB • zpg	ORA zpg, X	ASL zpg, X		CLC	ORA abs, Y	INC • A		TRB • abs	ORA abs, X	ASL abs, X		1
2	JSR abs	AND ind, X			BIT zpg	AND zpg	ROL zpg		PLP	AND imm	ROL A		BIT abs	AND abs	ROL abs		2
3	BMI rel	AND ind, Y	AND • ind		BIT • zpg, X	AND zpg, X	ROL zpg, X		SEC	AND abs, Y	DEC • A		BIT • abs, X	AND abs, X	ROL abs, X		3
4	RTI	EOR ind, X				EOR zpg	LSR zpg		PHA	EOR imm	LSR A		JMP abs	EOR abs	LSR abs		4
5	BVC rel	EOR ind, Y	EOR • ind			EOR zpg, X	LSR zpg, X		CLI	EOR abs, Y	PHY •			EOR abs, X	LSR abs, X		5
6	RTS	ADC ind, X			STZ • zpg	ADC zpg	ROR zpg		PLA	ADC imm	ROR A		JMP ind	ADC abs	ROR abs		6
7	BVS rel	ADC ind, Y	ADC • ind		STZ • zpg, X	ADC zpg, X	ROR zpg, X		SEI	ADC abs, Y	PLY •		JMP • ind, X	ADC abs, X	ROR abs, X		7
8	BRA • rel	STA ind, X			STY zpg	STA zpg	STX zpg		DEY	BIT • imm	TXA		STY abs	STA abs	STX abs		8
9	BCC rel	STA ind, Y	STA • ind		STY zpg, X	STA zpg, X	STX zpg, Y		TYA	STA abs, Y	TXS		STZ • abs	STA abs, X	STZ • abs, X		9
A	LDY imm	LDA ind, X	LDX imm		LDY zpg	LDA zpg	LDX zpg		TAY	LDA imm	TAX		LDY abs	LDA abs	LDX abs		A
B	BCS rel	LDA ind, Y	LDA • ind		LDY zpg, X	LDA zpg, X	LDX zpg, Y		CLV	LDA abs, Y	TSX		LDY abs, X	LDA abs, X	LDX abs, Y		B
C	CPY imm	CMP ind, X			CPY zpg	CMP zpg	DEC zpg		INY	CMP imm	DEX		CPY abs	CMP abs	DEC abs		C
D	BNE rel	CMP ind, Y	CMP • ind			CMP zpg, X	DEC zpg, X		CLD	CMP abs, Y	PHX •			CMP abs, X	DEC abs, X		D
E	CPX imm	SBC ind, X			CPX zpg	SBC zpg	INC zpg		INX	SBC imm	NOP		CPX abs	SBC abs	INC abs		E
F	BEQ rel	SBC ind, Y	SBC • ind			SBC zpg, X	INC zpg, X		SED	SBC abs, Y	PLX •			SBC abs, X	INC abs, X		F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Nota: • Nova Instrução

* Instrução Antiga com novo Modo de Endereçamento

Legendas da Matriz de Op Codes

MNEMÔNICO DA INSTRUÇÃO	(COMENTÁRIO)	MODO DE ENDEREÇAMENTO
Nº BÁSICO DE BYTES		Nº BÁSICO DE CICLOS

Nota: © Western Design Center, Inc.
Usado sob permissão

MODOS DE ENDEREÇAMENTO DO 65C02

São disponíveis quinze modos de endereçamento para a família W65SCXXX da Western Design Center:

Endereçamento Imediato
Neste modo, o operando está contido no segundo byte da instrução, não sendo necessários endereçamentos adicionais.

Endereçamento Absoluto
Para endereçamento absoluto, o segundo byte da instrução especifica os 8 bits de ordem inferior do endereço efetivo, enquanto o terceiro byte especifica os 8 bits de ordem superior. Assim, este modo possibilita endereçar 65K bytes de memória.

Endereçamento da Página Zero
Este modo permite códigos mais curtos e tempos de execução menores, por tomar somente o segundo byte da instrução e assumir zero para o byte superior do endereço, o que pode representar um aumento significativo na eficiência do código.

Endereçamento Implicado
Neste modo, o endereço que contém o operando é implicitamente estabelecido no código operacional da instrução.

Endereçamento do Acumulador
Esta forma de endereçamento é representada por uma instrução de um byte, implicitando uma operação no acumulador. Os op codes incluem-se no endereçamento implicado.

Endereçamento Indexado Indireto, Página Zero (IND,X)
Neste modo (usualmente denominado Indireto,X), o segundo byte da instrução é somado ao conteúdo do registro X; o transporte é desconsiderado. O resultado desta adição aponta uma posição de memória na Página Zero cujo conteúdo é o byte de ordem inferior do endereço efetivo. A próxima posição de memória da Página Zero contém o byte superior do endereço efetivo. Ambas as posições de memória especificando os bytes inferior e superior do endereço efetivo precisam pertencer à Página Zero.

Endereçamento Absoluto Indexado Indireto (Somente Instrução de "Jump")
Neste modo, o conteúdo do segundo e do terceiro bytes da instrução são somados ao registro X. O

resultado aponta uma posição de memória que contém os 8 bits de ordem inferior do endereço efetivo. A próxima posição de memória contém os 8 bits de ordem superior do endereço efetivo (op code 7C).

Endereçamento Indireto Indexado: (IND),Y
Esta forma de endereçamento é usualmente denominada endereçamento indireto, Y. O segundo byte da instrução aponta uma posição de memória na Página Zero. O conteúdo dessa posição é somado ao do registro Y, sendo o resultado o byte inferior do endereço efetivo. O transporte desta adição é somado ao conteúdo da próxima posição da Página Zero, sendo o resultado o byte de ordem superior do endereço efetivo.

Endereçamento Indexado, Página Zero
Este modo é usado em conjunto com os registros X ou Y e chamado "Página Zero, X" ou "Página Zero, Y". O endereço efetivo é calculado pela adição do segundo byte ao conteúdo do registro. Como esta é uma forma de endereçamento "Página Zero", o conteúdo do segundo byte referencia uma posição na Página Zero. Devido à natureza de endereçamento "Página Zero", não é adicionado transporte aos 8 bits de ordem superior, de forma que não ocorre extravasamento dos limites da página.

Endereçamento Absoluto Indexado
O endereçamento absoluto indexado é usado em conjunto com os registros X e Y, e a ele nos referimos como "Absoluto, X" ou "Absoluto, Y". O endereço efetivo é formado pela adição do conteúdo de X e Y ao endereço contido no segundo e terceiro bytes da instrução. Este modo permite que o registro contenha o valor índice, ou valor de contagem, e que a instrução contenha o endereço-base. Este tipo de indexação permite referenciar qualquer localização e ao índice modificar campos múltiplos, resultando em redução de código e de tempo de execução.

Endereçamento Relativo
O endereçamento relativo é usado somente com instruções de desvio; ele estabelece um destino para o desvio condicional.

Endereçamento Indireto, Página Zero: Indireto
Nesta forma de endereçamento, o segundo byte da instrução contém os 8 bits de ordem inferior de uma localização de memória. Os 8 bits de ordem superior são sempre zero. O conteúdo da posição de memória especificada é o byte de ordem inferior do endereço efetivo. A próxima posição de memória contém o byte de ordem superior do endereço efetivo.

Endereçamento Absoluto Indireto (Somente Instrução de "Jump")
O segundo byte da instrução contém os 8 bits de ordem inferior de uma posição de memória. Os 8 bits de ordem superior dessa posição de memória encontram-se no terceiro byte da instrução. O conteúdo da posição de memória assim especificada é o byte de ordem inferior do endereço efetivo. A próxima posição de memória contém o byte de ordem superior do endereço efetivo, que é carregado nos 16 bits do contador de programa (op code 6C).

Nota: © Western Design Center, Inc. Usado sob permissão

F

INSTRUÇÕES DO 65802/65816

A. O Jogo de Instruções Original do 6502 (151 Op Codes)

1.	ADC	Add Memory to Accumulator with Carry	Somar Memória ao Acumulador, com Transporte
2.	AND	"AND" Memory with Accumulator	"E" da Memória com o Acumulador
3.	ASL	Shift Left One Bit (Memory or Accumulator)	Deslocar um Bit à Esquerda (Memória ou Acumulador)
4.	BCC	Branch on Carry Clear	Desviar se o Transporte Estiver Desativado
5.	BCS	Branch on Carry Set	Desviar se o Transporte Estiver Ativado
6.	BEQ	Branch if Equal (Zero Flag Set)	Desviar se Igual (Indicador de Zero Ativado)
7.	BIT	Test Bits in Memory with Accumulator	Verificar Bits da Memória com os do Acumulador
8.	BMI	Branch on Result Minus	Desviar se o Resultado for Negativo
9.	BNE	Branch on Result Not Zero	Desviar se o Resultado for Diferente de Zero
10.	BPL	Branch on Result Plus	Desviar se o Resultado for Positivo
11.	BRK	Force Break	Parada Forçada
12.	BVC	Branch on Overflow Clear	Desviar se o "Overflow" Estiver Desativado
13.	BVS	Branch on Overflow Set	Desviar se o "Overflow" Estiver Ativado
14.	CLC	Clear Carry Flag	Desativar Indicador de Transpote
15.	CLD	Clear Decimal Mode	Desativar Modo Decimal
16.	CLI	Clear Interrupt Disable Bit	Desativar Bit de Desativamento de Interrupções
17.	CLV	Clear Overflow Flag	Desativar Indicador de "Overflow"
18.	CMP	Compare Memory and Accumulator	Comparar Memória com o Acumulador
19.	CPX	Compare Memory and Index X	Comparar Memória com o Índice X
20.	CPY	Compare Memory and Index Y	Comparar Memória com o Índice Y
21.	DEC	Decrement Memory by One	Decrementar Memória em uma Unidade
22.	DEX	Decrement Index X by One	Decrementar o Índice X em uma Unidade
23.	DEY	Decrement Index Y by One	Decrementar o Índice Y em uma Unidade
24.	EOR	"Exclusive-or" Memory with Accumulator	"OU Exclusivo" entre Memória e Acumulador
25.	INC	Increment Memory by One	Incrementar a Memória em uma Unidade
26.	INX	Increment Index X by One	Incrementar o Índice X em uma Unidade

27. INY	Increment Index Y by One	Incrementar o Índice Y em uma Unidade
28. JMP	Jump to New Location	Pular para uma Nova Posição
29. JSR	Jump to New Location Saving Return Address	Pular para uma Nova Posição, Guardando o Endereço de Retorno
30. LDA	Load Accumulator with Memory	Carregar o Acumulador com a Memória
31. LDX	Load Index X with Memory	Carregar o Índice X com a Memória
32. LDY	Load Index Y with Memory	Carregar o Índice Y com a Memória
33. LSR	Shift One Bit Right (Memory or Accumulator)	Deslocar um Bit à Direita (Memória ou Acumulador)
34. NOP	No Operation	Nenhuma Operação
35. ORA	"OR" Memory with Accumulator	"OU" entre Memória e Acumulador
36. PHA	Push Accumulator on Stack	Empurrar o Acumulador para a Pilha
37. PHP	Push Processor Status on Stack	Empurrar o Registro de Status do Processador para a Pilha
38. PLA	Pull Accumulator from Stack	Puxar da Pilha para o Acumulador
39. PLP	Pull Processor Status from Stack	Puxar da Pilha para o Registro de Status do Processador
40. ROL	Rotate One Bit Left (Memory or Accumulator)	Rotação de um Bit para a Esquerda (Memória ou Acumulador)
41. ROR	Rotate One Bit Right (Memory or Accumulator)	Rotação de um Bit para a Direita (Memória ou Acumulador)
42. RTI	Return from Interrupt	Retornar da Interrupção
43. RTS	Return from Subroutine	Retornar da Sub-rotina
44. SBC	Subtract Memory from Accumulator with Borrow	Subtrair Memória do Acumulador, com Empréstimo
45. SEC	Set Carry Flag	Ativar Indicador de Transporte
46. SED	Set Decimal Mode	Ativar Modo Decimal
47. SEI	Set Interrupt Disable Status	Ativar Desabilitação de Interrupções
48. STA	Store Accumulator in Memory	Armazenar o Acumulador na Memória
49. STX	Store Index X in Memory	Armazenar o Índice X na Memória
50. STY	Store Index Y in Memory	Armazenar o Índice Y na Memória
51. TAX	Transfer Accumulator to Index X	Transferir o Acumulador para o Índice X
52. TAY	Transfer Accumulator to Index Y	Transferir o Acumulador para o Índice Y
53. TSX	Transfer Stack Pointer to Index X	Transferir o Ponteiro da Pilha para o Índice X
54. TXA	Transfer Index X to Accumulator	Transferir o Índice X para o Acumulador
55. TXS	Transfer Index X to Stack Register	Transferir o Índice X para o Registro da Pilha
56. TYA	Transfer Index Y to Accumulator	Transferir o Índice Y para o Acumulador

B. Novas Instruções (13 Op Codes) do W65SCXXX

1. BRA	Branch Relative Always	Desviar Sempre
2. PLX	Pull X from Stack	Puxar X da Pilha
3. PLY	Pull Y from Stack	Puxar Y da Pilha
4. PHX	Push X on Stack	Empurrar X para a Pilha
5. PHY	Push Y on Stack	Empurrar Y para a Pilha
6. STZ	Store Zero in Memory (Direct; Direct, X; Abs; Abs,X)	Armazenar Zero na Memória (Direto; Direto,X; Abs; Abs,X)
7. TRB	Test and Reset Memory Bits Determined by the Accumulator A (Direct and Absolute)	Verificar e Desativar Bits da Memória, Determinados pelo Acumulador A (Direto e Absoluto)
8. TSB	Test and Set Memory Bits Determined by the Accumulator A (Direct and Absolute)	Verificar e Ativar Bits da Memória, Determindos pelo Acumulador A (Direto e Absoluto)

C. Novos Modos de Endereçamento (14 Op Codes) do W65SCXXX

1. BIT	Test Bits in Memory with Accumulator (Direct,X; Absolute,X; Immediate)	Verificar Bits da Memória com os do Acumulador (Direto,X; Absoluto,X; Imediato)
2. DEC	Decrement (Accumulator)	Decrementar (Acumulador)
3. Group 1	Instructions (Diret Indirect (8 Op Codes))	Instruções (Direto Indireto (8 Op Codes))
4. INC	Increment (Accumulator)	Incrementar (Acumulador)
5. JMP	Jump to New Location (Absolute Indexed Indirect)	Pular para uma Nova Posição (Absoluto Indexado Indireto)

D. Instruções do Grupo 1 com Novos Modos de Endereçamento (48 Op Codes)

- Direto Indireto Longo Indexado com Y (8 Op Codes)
- Direto Indireto Longo (8 Op Codes)
- Absoluto Longo e Absoluto Longo Indexado com X (16 Op Codes)
- Pilha, Relativo (8 Op Codes)

1. ADC	Add Memory to Accumulator with Carry	Somar Memória ao Acumulador, com Transporte
2. AND	"AND" Memory with Accumulator	"E" da Memória com o Acumulador
3. CMP	Compare Memory and Accumulator	Comparar Memória com o Acumulador
4. EOR	"Exclusive-or" Memory with Accumulator	"OU Exclusivo" entre Memória e Acumulador
5. LDA	Load Accumulator with Memory	Carregar Acumulador com a Memória
6. ORA	"OR" Memory with Accumulator	"OU" entre Memória e Acumulador
7. SBC	Subtract Memory from Accumulator with Borrow	Subtrair Memória do Acumulador, com Empréstimo
8. STA	Store Accumulator in Memory	Armazenar o Acumulador na Memória

E. Novas Instruções "Puxar e Empurrar" (7 Op Codes)

1. PEA	Push Effective Absolute Address or Immediate Data Word on Stack	Empurrar Endereço Absoluto Efetivo ou Palavra de Dados Imediata para a Pilha
2. PEI	Push Effective Indirect Address or Direct Data Word on Stack	Empurrar Endereço Indireto Efetivo ou Palavra de Dados Direta para a Pilha
3. PER	Push Effective Program Counter Relative Indirect Address or Program Counter Relative Data Word on Stack	Empurrar o Endereço Indireto Relativo Efetivo do Contador de Programa, ou a Palavra de Dados Relativa do Contador de Programa para a Pilha
4. PLB	Pull Data Bank Register from Stack	Puxar o Registro de Banco de Dados da Pilha
5. PLD	Pull Direct Register from Stack	Puxar o Registro Direto da Pilha
6. PHB	Push Data Bank Register on Stack	Empurrar o Registro de Banco de Dados para a Pilha
7. PHD	Push Direct Register on Stack	Empurrar o Registro Direto para a Pilha
8. PHK	Push Program Bank Register on Stack	Empurrar o Registro de Banco do Programa para a Pilha

F. Instruções do Registro de Status (2 Op Codes)

1. REP	Reset Status Bits Defined by Immediate Byte 1 = Reset 0 = Do not change	Desativar Bits de Status, definidos por: Byte Imediato 1 = Desativar 0 = Não alterar
2. SEP	Set Status Bits Defined by Immediate Byte 1 = Set 0 = Do not change	Ativar Bits de Status, Definidos por: Byte Imediato 1 = Ativar 0 = Não alterar

G. Novas Instruções de Transferência de Registros (8 Op Codes)

1. TCD	Transfer C Accumulator to Direct Register D	Transferir o Acumulador C para o Registro Direto D
2. TDC	Transfer Direct Register D to C Accumulator	Transferir o Registro Direto D para o Acumulador C
3. TCS	Transfer C Accumulator to Stack Register	Transferir o Acumulador C para o Registro da Pilha
4. TSC	Transfer Stack Register to Accumulator C	Transferir o Registro da Pilha para o Acumulador C
5. TXY	Transfer X to Y	Transferir X para Y
6. TYX	Transfer Y to X	Transferir Y para X
7. XBA	Exchange Band A	Intercambiar B e A
8. SCE	Exchange Carry Bit C with Emulation Bit E	Intercambiar o Bit de Transporte C e o Bit de Emulação E

H. Novas Instruções de Desvio, "Jump" e Retorno (6 Op Codes)

1. BRL	Branch Relative Long Always (16 Bit Relative – 32768 to + 32767) (Addressing Mode)	Desvio Relativo Longo Sempre (16 Bit Relative – 32768 a + 32767) (Modo de Endereçamento)
2. JML	Jump Indirect Long	Salto Indireto Longo
3. JMP	Jump Absolute Long	Salto Absoluto Longo
4. JSL	Jump to Subroutine Long (Uses RTL for Return)	Salto para Sub-rotina Longo (Usar RTL para Retorno)
5. JSR	Jump to Subroutine (Indexed Indirect)	Salto para Sub-rotina (Indexado Indireto)
6. RTL	Return from Subroutine Long	Retorno de Sub-rotina Longo

I. Novas Instruções de Movimentação de Blocos (2 Op Codes)

1. MVN	Move Block from Source (X Addressed) to Destination (Y Addressed), Block Length Defined by C, X, Y are Incremented	Mover Bloco da Origem (Endereçamento X) para o Destino (Endereçamento Y), Comprimento do Bloco Definido por C, X e Y são Incrementados
2. MVP	Move Block from Source (X Addressed) to Destination (Y Addressed), Block Length Defined by C, X, Y are Decremented	Mover Bloco da Origem (Endereçamento X) para o Destino (Endereçamento Y), Comprimento do Bloco Definido por C, X e Y são Decrementados

J. Novas Operações de Co-Processador (1 Op Code)

1. COP	Co-Processor Instruction with Associated COP Vector and ABORT Input Supports Co-Processing Function, i.e., Floating Point Processors etc.	Instrução para Co-processador, com um Vetor COP Associado e com uma Entrada ABORT, Suportando Funções de Co-processamento; por exemplo, Processadores em Ponto Flutuante etc.

K. Novas Instruções de Controle do Sistema (3 Op Codes)

1.	Stop-the-clock Instruction Stops the Oscillator Input (or 02 input) During 02 = 1. This Mode is Released when RES Goes to a Zero. System Initialization May Be Desired; However, if After RESET One Performed an RTI, Program Execution Begins with the Instruction Following the STP Op Code in Program Sequence	Esta Instrução ("stop the clock") Pára o Oscilador Enquanto a Entrada 02 for mantida em 1. Este Modo é Desativado quando RES vai para Zero. Pode-se efetuar a Inicialização do Sistema; Entretanto, se após o RESET for executado um RTI, o processamento do Programa começará na Instrução que segue o Op Code STP na Seqüência do Programa
2. WAI	Wait for Interrupt, Pulls RDY Low and Is Cleared by IRQ or NMI Active Input	Aguardar por Interrupção faz com que RDY fique baixo e fica desativado pelo IRQ ou uma entrada através do NMI
2. WDM	Reserved Code for Future Use W65SC816 Performs A No-Operation	Código Reservado para uso Futuro O W65SC816 executa uma Não-Operação.

Nota: © Western Design Center, Inc
Usado sob permissão

G

MODOS DE ENDEREÇAMENTO DO 65816

Modos de Endereçamento
São disponíveis vinte e quatro modos de endereçamento na família W65SC816 de microprocessadores.

1. Endereçamento Imediato [imm]
No endereçamento imediato, o operando está contido no segundo byte (segundo e terceiro bytes para dados em 16 bits) da instrução.

2, 3. Endereçamentos Absoluto e Absoluto Longo [a], [al]
Para endereçamento absoluto, o segundo byte da instrução especifica os 8 bits de ordem inferior do endereço efetivo, enquanto o terceiro byte especifica os 8 bits de ordem superior. Para endereçamento absoluto longo, o quarto byte especifica o endereço do banco. Todo o espaço de 16,7 Megabytes é endereçado em modo longo. No modo *short*, o endereço do banco é especificado pelo registro do banco de dados.

4. Endereçamento Direto [d]
Este modo permite utilizar menos código e obter tempos de execução menores, por tomar somente o segundo byte da instrução. O segundo byte é somado ao valor do registro direto (D). Quando o registro direto inferior (DL) é zero, a execução ocorre da forma mais rápida. O endereço de banco é sempre zero.

5. Endereçamento do Acumulador [acc]
Esta forma de endereçamento é representada com uma instrução de um byte e realiza uma operação no(s) acumulador(es).

6. Endereçamento Implicado [imp]
Neste modo, o endereço do operando é, implicitamente, estabelecido no código de operação da instrução.

7, 8. Endereçamentos Direto Indireto Indexado e Direto Indireto Indexado Longo [(d), y] [(dl), y]
Este tipo de endereçamento é usualmente denominado Indireto,Y. O segundo byte da instrução é somado ao registro direto e aponta para uma posição de memória no banco zero. O conteúdo desta localização de memória e o byte seguinte (o próximo byte é o endereço do banco no modo longo) são somados ao registro Y, sendo o resultado o endereço efetivo. No modo *short*, o endereço do banco é especificado pelo registro de banco de dados. Quando DL é zero, a execução é a mais rápida.

9. Endereçamento Indireto Indexato Direto [(d,x)]
Neste modo, também chamado de Indireto,X, o segundo byte da instrução é somado ao conteúdo do registro direto e então se adiciona o valor do registro X. O resultado destas adições aponta uma posição de memória, no banco zero, cujo conteúdo é o byte de ordem inferior do endereço efetivo e o byte seguinte o byte de ordem superior. O endereço do banco do endereço efetivo é especificado pelo registro de banco de dados.

10, 11 Endereçamentos Direto Indexado com X e Direto Indexado com Y [d,x], [d,y]
Também chamados de Direto,X e Direto,Y, estes endereçamentos são instruções de dois bytes. O segundo byte é somado ao registro direto (D) e o resultado é somado ao registro-índice (X ou Y) adequado. O endereço do banco é sempre zero. A execução é a mais rápida quando o byte inferior (DL) do registro direto é zero.

12, 13, 14. Endereçamentos Absoluto Indexado com X, Absoluto Indexado Longo com X e Absoluto Indexado com Y [a,x], [al,x], [a,y]
O endereçamento absoluto indexado é usado em conjunto com os registros X e Y e é chamado de Absoluto,X, Absoluto Longo,X e Absoluto,Y. O endereço efetivo é formado pela adição do conteúdo do registro X ou Y ao segundo e terceiro bytes da instrução. O endereço do banco é especificado pelo registro de banco de dados, exceto no modo longo, onde o quarto byte especifica o endereço do banco.

15, 16. Endereçamentos do Contador de Programa, Relativo e Relativo Longo [r], [rl]
Também chamado de relativo e relativo longo, o endereçamento relativo do contador de programa é usado somente com instruções de desvio. O segundo byte é somado ao contador de programa que, por relação, cria um deslocamento de +128 ou –127 bytes. O segundo e terceiro bytes são adicionados ao contador para criar um deslocamento de +32768 ou –32767 bytes para as operações de desvio longo.

17. Endereçamento Absoluto Indireto (somente Instruções de "Jump") [(a)]
O segundo e terceiro bytes da instrução contêm os bytes inferior e superior de uma posição de memória localizada no banco zero. Esta posição e o byte seguinte contêm o endereço efetivo a ser carregado no contador de programa. O endereço do banco de destino é especificado pelo registro de bancos do programa, exceto no caso da instrução JML, onde o terceiro byte especifica o endereço do banco de destino.

18, 19. Endereçamentos Direto Indireto e Direto Indireto Longo [(d)], [(dl)]
Neste modo, o segundo byte da instrução é somado ao registro direto, e o resultado aponta uma posição de memória no banco zero. O conteúdo desta posição e o da seguinte (a próxima posição é o endereço do banco para o modo longo) formam o endereço efetivo. O endereço do banco é especificado pelo registro de banco de dados, no modo direto indireto.

20. Endereçamento Absoluto Indexado Indireto (Jump e Jump para Sub-rotina) [(a,x)]
Neste modo, o segundo e o terceiro bytes da instrução são somados ao conteúdo do registro X. O resultado aponta o byte inferior e o superior (byte seguinte) que são carregados no contador de programa. O endereço do banco é especificado pelo registro de bancos do programa.

21. Endereçamento da Pilha [s]
Este modo utiliza o registro da pilha para endereçar posições na memória. As instruções que usam

endereçamento da pilha incluem puxar, empurrar, interrupções, salto para sub-rotina, retorno de interrupção e retorno de sub-rotina. O endereço do banco é sempre zero. Vetores são sempre puxados do banco 00. (Ver Compatibility Issues for 6502 Emulation.)

22. Endereçamento Relativo da Pilha [sr]
O segundo byte da instrução é somado ao valor do registro da pilha. Este endereço efetivo aponta uma posição de memória de dados na pilha. Para dados em 16 bits, a próxima posição na pilha é o byte superior de dados. Este modo, em conjunto com instruções "empurrar", pode ser usado para passar dados a sub-rotinas através da pilha. As novas instruções TSC e TCS permitem modificações rápidas na pilha. O registro direto pode ser usado para funções de pilha. O registro de banco é sempre zero.

23. Endereçamento Relativo Indireto Indexado da Pilha [(sr),y]
No endereçamento relativo indireto indexado em Y, o segundo byte da instrução é somado ao valor do registro da pilha. O endereço formado por essa adição aponta o byte inferior (a próxima posição contém o byte superior) de um endereço indireto. O registro Y é somado a esse endereço para formar o endereço efetivo dos dados. Este modo, em conjunto com o uso de endereços efetivos de instruções "push" (PEA, PEI, PER), pode ser usado para passar dados para sub-rotinas, utilizando a pilha. As novas instruções TSC e TCS permitem modificar rapidamente a pilha. O registro direto pode ser usado para funções da pilha. O registro de banco de dados é o endereço do banco para o endereço efetivo.

24. Endereçamento de Movimentação de Blocos [xyc]
Este modo de endereçamento é usado para efetuar a movimentação de bytes múltiplos para frente (MVP) e para trás (MVN). Estas instruções, de três bytes, usam o registro X como endereço origem e o registro Y como endereço de destino; o acumulador C contém o número de bytes a serem movidos. O endereço do banco de destino é o segundo byte da instrução, e o banco de origem é especificado pelo terceiro byte. O registro de banco de dados é carregado com o valor do banco de destino (segundo byte da instrução).

Nota: © Western Design Center, Inc.
Usado sob permissão

TABELA DE OP CODES DO 65816

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s 2 8	ORA(d,x) 2 6	COP s 2*8	ORA sr 2*4	TSB d 2*5	ORA d 2 3	ASL d 2 5	ORA[d] 2*6	PHP s 1 3	ORA imm 2 2	ASL acc 1 2	PHD s 1*4	TSB a 3*6	ORA a 3 4	ASL a 3 6	ORA al 4*5	0
1	BPL r 2 2	ORA(d),y 2 5	ORA (d) 2*5	ORA(sr),y 2*7	TRB d 2*5	ORA d,x 2 4	ASL d,x 2 6	ORA[d],y 2*6	CLC imp 1 2	ORA a,y 3 4	INC acc 1*2	TCS imp 1*2	TRB a 3*6	ORA a,x 3 4	ASL a,x 3 7	ORA al,x 4*5	1
2	JSR a 3 6	AND(d,x) 2 6	JSL al 4*8	AND sr 2*4	BIT d 2 3	AND d 2 3	ROL d 2 5	AND[d] 2*6	PLP s 1 4	AND imm 2 2	ROL acc 1 2	PLD s 1*5	BIT a 3 4	AND a 3 4	ROL a 3 6	AND al 4*5	2
3	BMI r 2 2	AND(d),y 2 5	AND(d) 2*5	AND(sr),y 2*7	BIT d,x 2*4	AND d,x 2 4	ROL d,x 2 6	AND[d],y 2*6	SEC imp 1 2	AND a,y 3 4	DEC acc 1*2	TSC imp 1*2	BIT a,x 3*4	AND a,x 3 4	ROL a,x 3 7	AND al,x 4*5	3
4	RTI s 1 7	EOR(d,x) 2 6	WDM RESERVED	EOR sr 2*4	MVP xyc 3*7	EOR d 2 3	LSR d 2 5	EOR[d] 2*6	PHA s 1 3	EOR imm 2 2	LSR acc 1 2	PHK s 1*3	JMP a 3 3	EOR a 3 4	LSR a 3 6	EOR al 4*5	4
5	BVC r 2 2	EOR(d),y 2 5	EOR (d) 2*5	EOR(sr),y 2*7	MVN xyc 3*7	EOR d,x 2 4	LSR d,x 2 6	EOR[d],y 2*6	CLI imp 1 2	EOR a,y 3 4	PHY s 1*3	TCD imp 1*2	JMP al 3*4	EOR a,x 3 4	LSR a,x 3 7	EOR al,x 4*5	5
6	RTS s 1 6	ADC(d,x) 2 6	PER s 3*6	ADC sr 2*4	STZ d 2*3	ADC d 2 3	ROR d 2 5	ADC[d] 2*6	PLA s 1 4	ADC imm 2 2	ROR acc 1 2	RTL s 1*6	JMP (a) 3 5	ADC a 3 4	ROR a 3 6	ADC al 4*5	6
7	BVS r 2 2	ADC(d),y 2 5	ADC(d) 2*5	ADC(sr),y 2*7	STZ d,x 2*4	ADC d,x 2 4	ROR d,x 2 6	ADC[d],y 2*6	SEI imp 1 2	ADC a,y 3 4	PLY s 1*4	TDC imp 1*2	JMP(a,x) 3*6	ADC a,x 3 4	ROR a,x 3 7	ADC al,x 4*5	7
8	BRA r 2*2	STA(d,x) 2 6	BRL rl 3*3	STA sr 2*4	STY d 2 3	STA d 2 3	STX d 2 3	STA[d] 2*6	DEY imp 1 2	BIT imm 2*2	TXA imm 1 2	PHB s 1*3	STY a 3 4	STA a 3 4	STX a 3 6	STA al 4*5	8
9	BCC r 2 2	STA(d),y 2 6	STA (d) 2*5	STA(sr),y 2*7	STY d,x 2 4	STA d,x 2 4	STX d,y 2 4	STA[d],y 2*6	TYA imp 1 2	STA a,y 3 5	TXS imp 1 2	TXY imp 1*2	STZ a 3*4	STA a,x 3 5	STZ a,x 3*5	STA al,x 4*5	9
A	LDY imm 2 2	LDA(d,x) 2 6	LDX imm 2 2	LDA sr 2*4	LDY d 2 3	LDA d 2 3	LDX d 2 3	LDA[d] 2*6	TAY imp 1 2	LDA imm 2 2	TAX imp 1 2	PLB s 1*4	LDY a 3 4	LDA a 3 4	LDX a 3 4	LDA al 4*5	A
B	BCS r 2 2	LDA(d),y 2 5	LDA(d) 2*5	LDA(sr),y 2*7	LDY d,x 2 4	LDA d,x 2 4	LDX d,y 2 4	LDA[d],y 2*6	CLV imp 1 2	LDA a,y 3 4	TSX imp 1 2	TYX imp 1*2	LDY a,x 3 4	LDA a,x 3 4	LDX a,y 3 4	LDA al,x 4*5	B
C	CPY imm 2 2	CMP(d,x) 2 6	REP imm 2*3	CMP sr 2*4	CPY d 2 3	CMP d 2 3	DEC d 2 5	CMP[d] 2*6	INY imp 1 2	CMP imm 2 2	DEX imm 1 2	WAI imp 1*3	CPY a 3 4	CMP a 3 4	DEC a 3 6	CMP al 4*5	C
D	BNE r 2 2	CMP(d),y 2 5	CMP (d) 2*5	CMP(sr),y 2*7	PEI s 2*6	CMP d,x 2 4	DEC d,x 2 6	CMP[d],y 2*6	CLD imp 1 2	CMP a,y 3 4	PHX s 1*3	STP imp 1*3	JML (a) 3*6	CMP a,x 3 4	DEC a,x 3 7	CMP al,x 4*5	D
E	CPX imm 2 2	SBC(d,x) 2 6	SEP imm 2*3	SBC sr 2*4	LPX d 2 3	SBC d 2 3	INC d 2 5	SBC[d] 2 6	INX imp 1 2	SBC imm 2 2	NOP imp 1 2	XBA imp 1*3	CPX a 3 4	SBC a 3 4	INC a 3 6	SBC al 4*5	E
F	BEQ r 2 2	SBC(d),y 2 5	SBC (d) 2*5	SBC(sr),y 2*7	PEA s 3*5	SBC d,x 2 4	INC d,x 2 6	SBC[d],y 2*6	SED imp 1 2	SBC a,y 3 4	PLX s 1*4	XCE imp 1*2	JSR(a,x) 3*6	SBC a,x 3 4	INC a,x 3 7	SBC al,x 4*5	F

* Novos Op Codes do W65SC816
• Op Codes do W65SC02

Legenda para a Matriz de Op Codes



Nota: © Western Design Center, Inc.
Usado com permissão

I

CONJUNTO DE CARACTERES ASCII PARA O APPLE II

A. $00-3F: Caracteres em Vídeo Inverso

Hexa	Dec	Tela
$00	0	@
$01	1	A
$02	2	B
$03	3	C
$04	4	D
$05	5	E
$06	6	F
$07	7	G
$08	8	H
$09	9	I
$0A	10	J
$0B	11	K
$0C	12	L
$0D	13	M
$0E	14	N
$0F	15	O
$10	16	P
$11	17	Q
$12	18	R
$13	19	S
$14	20	T
$15	21	U
$16	22	V
$17	23	W
$18	24	X
$19	25	Y
$1A	26	Z
$1B	27	[
$1C	28	\
$1D	29	]

Hexa	Dec	Tela
$1E	30	'
$1F	31	-
$20	32	ESPAÇO
$21	33	!
$22	34	"
$23	35	#
$24	36	$
$25	37	%
$26	38	&
$27	39	'
$28	40	(
$29	41	)
$2A	42	*
$2B	43	+
$2C	44	,
$2D	45	-
$2E	46	.
$2F	47	/
$30	48	0
$31	49	1
$32	50	2
$33	51	3
$34	52	4
$35	53	5
$36	54	6
$37	55	7
$38	56	8
$39	57	9
$3A	58	:
$3B	59	;
$3C	60	<
$3D	61	=
$3E	62	>
$3F	63	?

B. $40-7F: Caracteres Piscantes*

Hexa	Dec	Tela
$40	64	@
$41	65	A
$42	66	B
$43	67	C
$44	68	D
$45	69	E

* Os caracteres ASCII de $40 a $5F são apresentados como caracteres especiais MouseText se o firmware do mouse estiver instalado e ativo.

Hexa	Dec	Tela
$46	70	F
$47	71	G
$48	72	H
$49	73	I
$4A	74	J
$4B	75	K
$4C	76	L
$4D	77	M
$4E	78	N
$4F	79	O
$50	80	P
$51	81	Q
$52	82	R
$53	83	S
$54	84	T
$55	85	U
$56	86	V
$57	87	W
$58	88	X
$59	89	Y
$5A	90	Z
$5B	91	[
$5C	92	\
$5D	93	]
$5E	94	,
$5F	95	_
$60	96	ESPAÇO
$61	97	!
$62	98	"
$63	99	#
$64	100	$
$65	101	%
$66	102	&
$67	103	'
$68	104	(
$69	105	)
$6A	106	*
$6B	107	+
$6C	108	,
$6D	109	-
$6E	110	.
$6F	111	/
$70	112	0
$71	113	1
$72	114	2
$73	115	3
$74	116	4
$75	117	5
$76	118	6
$77	119	7

Hexa	Dec	Tela
$78	120	8
$79	121	9
$7A	122	:
$7B	123	;
$7C	124	<
$7D	125	=
$7E	126	>
$7F	127	?

C. $80-9F: Caracteres Control

Hexa	Dec	Tecla
$80	128	@
$81	129	A
$82	130	B
$83	131	C
$84	132	D
$85	133	E
$86	134	F
$87	135	G
$88	136	H
$89	137	I
$8A	138	J
$8B	139	K
$8C	140	L
$8D	141	M
$8E	142	N
$8F	143	O
$90	144	P
$91	145	Q
$92	146	R
$93	147	S
$94	148	T
$95	149	U
$96	150	V
$97	151	W
$98	152	X
$99	153	Y
$9A	154	Z
$9B	155	[
$9C	156	\
$9D	157	]
$9E	158	^
$9F	159	_

D. $A0-FF: Caracteres Normais

Hexa	Dec	Tecla
$A0	160	SPC

Hexa	Dec	Tecla
$A1	161	!
$A2	162	"
$A3	163	#
$A4	164	$
$A5	165	%
$A6	166	&
$A7	167	'
$A8	168	(
$A9	169	)
$AA	170	*
$AB	171	+
$AC	172	,
$AD	173	-
$AE	174	>
$AF	175	/
$B0	176	0
$B1	177	1
$B2	178	2
$B3	179	3
$B4	180	4
$B5	181	5
$B6	182	6
$B7	183	7
$B8	184	8
$B9	185	9
$BA	186	:
$BB	187	;
$BC	188	<
$BD	189	=
$BE	190	>
$BF	191	?
$C0	192	@
$C1	193	A
$C2	194	B
$C3	195	C
$C4	196	D
$C5	197	E
$C6	198	F
$C7	199	G
$C8	200	H
$C9	201	I
$CA	202	J
$CB	203	K
$CC	204	L
$CD	205	M
$CE	206	N
$CF	207	O
$D0	208	P
$D1	209	Q
$D2	210	R

	Dec	Tecla
$D3	211	S
$D4	212	T
$D5	213	U
$D6	214	V
$D7	215	W
$D8	216	X
$D9	217	Y
$DA	218	Z
$DB	219	[
$DC	220	\
$DD	221	]
$DE	222	'
$DF	223	–
$E0	224	,
$E1	225	a
$E2	226	b
$E3	227	c
$E4	228	d
$E5	229	e
$E6	230	f
$E7	231	g
$E8	232	h
$E9	233	i
$EA	234	j
$EB	235	k
$EC	236	l
$ED	237	m
$EE	238	n
$EF	239	o
$F0	240	p
$F1	241	q
$F2	242	r
$F3	243	s
$F4	244	t
$F5	245	u
$F6	246	v
$F7	247	w
$F8	248	x
$F9	249	y
$FA	250	z
$FB	251	[
$FC	252	/
$FD	253	]
$FE	254	'
$FF	255	Rubout

BIBLIOGRAFIA

Andrews, Mark. *Atari Roots: A Guide to Atari Assembly Language*. Chatsworth, CA: Datamost, 1984.

——— *The Apple II User's Guide*. New York: Macmillan, 1983.

——— *Programming the Commodore 64/128 in Assembly Language*. Indianopolis, IN: Howard W. Sams & Co., Inc., 1985.

Apple II Reference Manual (For IIe Only). Cupertino, CA: Apple Computer, Inc. 1982.

The Apple IIc Reference Manual. Cupertino, CA: Apple Computer, Inc., 1984.

Apple Mouse IIc User's Manual. Cupertino, CA: Apple Computer, Inc., 1984.

AppleMouse II User's Manual (For the Apple IIe, II Plus, and II). Cupertino, CA: Apple Computer, Inc., 1983.

Extended 80-Column Text/AppleColor Adaptor Card Manual. Cupertino, CA: Apple Computer, Inc., 1984.

Findley, Robert. *6502 Software Gourmet Guide & Cookbook*. Rochelle Park, NJ: Hayden Book Co., Inc., 1979.

Leventhal, Lance A. *6502 Assembly Language Programming*. Berkeley, CA: Osborne/McGraw-Hill, 1979.

Little, Gary B. *Inside the Apple IIe*. Bowie, MD: Brady Communications Co., Inc., 1985.

Maurer, W. Douglas. *Apple Assembly Language*. Rockville, MD: Computer Science Press, Inc., 1984.

Pelczarski, Mark. *Graphically Speaking*. Geneva, IL: Softalk Books, 1983.

ProDOS Assembler Tools. Cupertino, CA: Apple Computer, Inc., 1984.

ProDOS Technical Reference Manual for the Apple II Family. Cupertino, CA: Apple Computer, Inc., 1983.

Wagner, Roger. *Assembly Lines: The Book: A Beginner's Guide to 6502 Programming on the Apple II*. Santee, CA: Roger Wagner Publishing, Inc., 1984.

Zaks, Rodnay. *Programming the 6502*. Berkeley, CA: Sybex, 1983.



MARCAS REGISTRADAS

Os seguintes nomes são marcas registradas de produtos e seus respectivos fabricantes.

Apple®	Apple Computer, Inc.
Atari®	Warner Communications
Commodore 64™	Commodore Business Machines
The Complete Graphics System™	Penguin Software
CP/M®	Digital Research
The Graphics Magician™	Penguin Software
IBM®	International Business Machines, Inc.
Macintosh™	Apple Computer, Inc.*
Merlin Pro™	Roger Wagner Publishing, Inc.
ORCA/M®	The Byte Works, Inc.
Ping Pong®	Harvard Table Tennis
ProDOS®	Apple Computer, Inc.
Radio Shack®	Radio Shack, A Division of Tandy Corp.
Sourceror™	Roger Wagner Publishing, Inc.
Texas Instruments®	Texas Instruments, Inc.

* Macintosh é marca registada da Macintosh Laboratory, Inc., e a Apple tem licença para utilizá-la.



ÍNDICE ANALÍTICO

Abreviaturas usadas nos jogos de instruções, 105
Absoluto, endereçamento, 139
Absoluto indexado, endereçamento, 143, 308
Absoluto indexado indireto, endereçamento, 147
Absoluto indireto, endereçamento, 146
Absoluto, modo, 137
Acumulador, 41
  de ponto flutuante, 101
  endereçamento, 140
ADC, 106
Adição binária com sinal, 205
Adição de números, 196
ALU, 14, 40
AND, 106
Append, 58
Apple ProDOS Assembler, 53-71
Apple ProDOS Assembler Tools, VII-X, 53
Aritmética de ponto flutuante, 101
Arquitetura do Apple, 3-6
Arquitetura do chip, 49-52
ASCII, Código, 14
ASL, 107
Assembler, 2, 9
Assembler, ORCA/M, 83
Assembler, ProDOS, 68
Assembly, compilando um programa em, 53-83
Assembly, instruções em, 104-133
Assembly, linguagem, 8-11
Assembly, loops em, 167-171
Assembly, matemática, 193-210
Assembly, programa em, 35
Assembly, tabela de conversão para linguagem de máquina, 309-315

Bancos, chaveamento de, 39-40, 211, 213-214
BASIC, 1, 7-8, 15, 54, 56, 86, 95, 221, 268
BCC, 107
BCD, números, 65, 208-209
BCS, 107
BEQ, 108
Bibliografia, 343-344
BIT, 108
BIT, Operador, 190
Bit, 20
Bit descrição, 44-49
Bit de transporte, 178, 193-195
Bit, mapeamento, 270-271
Bits, deslocamento de, 179
BLOAD, 85
BMI, 108
BNE, 109
Booleana, lógica, 184
BPL, 110
BRA, 110
Break Flag, 47
BRK, 110
BRUN, 5, 85, 93
Buffer, 170
BVC, 111
BVS, 11

Byte, 3, 20
Byte Inferior, 176
Byte Superior, 176

CALL, comando em BASIC, 96
Campo Comentários, 64
Campo Op Code, 63
Campo Operando, 64
Campo "títulos", 62
Campos, 61
Carregando um programa em linguagem de máquina, 86-87
Chips, Apple, 6, 33-52
CLC, 112
CLD, 112
CLI, 112
CLV, 113
CMP, 113
Código Executável, 6
Código-fonte, 2, 12
Código-objeto, 2, 12
Código-objeto, supressão de, 71
Código Operacional, 63
Códigos de cores, 238
Códigos de cores da memória auxiliar, 241
Colunas, 61
Comentários no ORCA/M, 83
Comparação e desvio, 160
Comparando valores, 160
Compiladores, 7-8
Complemento da adição, 205-207
Comprimindo dados na memória, 187-191
Computadores de 8 bits, 36
Condicional, desvio, 160, 166
Contador de programa, 37-38, 42
Controles manuais, 246-267
Conversão de números, 23
CPX, 113
CPY, 113
Cores, 238

D, Registro, 52
Dados, 36
DEA, 114
DEC, 114
Decimal, indicador de modo, 46
Decimal, sistema, 19-31
Decrementar registros, 159
DELETE, 59
Descomprimindo dados da memória, 187-191
Deslocamento aritmético à esquerda (ASL), 179
Deslocamento lógico à direita (LSR), 180
Deslocamento (offset), 143, 163, 244
Deslocamento, valores de, 164-166
Desvio a longa distância, 165
Desvio condicional, 161, 166
Desvio e comparação, 160
Desvios, 142-143, 161-162
DEX, 114
DEY, 114
Diretivas, 158
Diretivas, ORCA/M, 82
Dirigindo listagens, 70
Divisão, 202-204
DOS 3.3, VII-VIII
Dupla leitura, operações de, 215-217

E, Indicador, 49
Edição, 58
Editor, 2, 58
Editor, ORCA/M, 81
Emulação, indicador, 49
Endereçamento, 105, 135-155
Endereçamento, modos de, 105-133, 135-156, 219
Endereçamento, modos de, 65C02, 325
Endereçamento, modos de, 65816, 332-342
Endereço de início, 34
Endereços, 36,243
Endereços, mapa de, 218-227
Endereços, modificação de, 307
Entrada e saída, 4
Entrada e saída, dispositivos de, 4, 37
EOR, 115
EOR, Operador, 186
Executando um programa em linguagem de máquina, 87

Figuras, 17

Game paddles, 246-268
GETLN1, 175
Gráfico e texto, modos, 233
Gráficos, 233-246

alta resolução, 238-240
baixa resolução, 237-238
dupla alta resolução, 242, 297-307
dupla baixa resolução, 240-242
mapeamento de tela, 243-245
nos Apple, 268-308
Programas, 273-292
Gravando um programa, 67
Merlin, 77
ORCA/M, 82

Hexadecimal, sistema, 10
Hexadecimais, números, 13, 19-31

Imediato, endereçamento, 138
Imediato, modo, 137
Implícito, endereçamento, 138
Impressão
Merlin, 77
ORCA/M, 82
Imprimindo em programa, 67
INA, 115
INC, 115
Incondicional, 162
Incrementando registros, 159
Indexado indireto, endereçamento, 147
Indicador de break, 47
Indireto, endereçamento, 146
Indireto indexado, endereçamento, 147
Inserir, 59
INSERT, 59
Instruções de salto, 161
Interpretadores, 7-8
Interrupção, 46
Interrupção, indicador, 46
Interrupção mascarável, 46
Interrupção não-mascarável, 46
INX, 115
INY, 116
I/O, 4
I/O, dispositivos, 37

JMP, 116
Jogo de caracteres alternativo, 17
Jogo de,
instruções 65C02, 321
instruções, 65802/65816, 327
instruções, 6502B/65C02, 104-133
Jogos, I/O, 246
Joysticks, 247-267
JSR, 116
"Jump", 161

KILL2, 60
Kilobyte ("K"), 211

LDA, 116
LDX, 116
LDY, 117
Linguagem de máquina, 1, 8-10
e linguagem assembly, 1-3, 317-320
programas, 5, 34, 85
Linha de origem, 64
Linha, números de, 56-58
Linhas, ORCA/M, 81
LIST, 58
Listagem, Merlin, 76
Lógicos, operadores, 184-188
"LOOPS", 156-178
LSI, 3
LSR, 117

Mapa de tela, 270
Mascarável, interrupção, 46
Matemática, 193-210
Memória, 3-6, 211-232
Apple, 34-40
Apple ProDOS Assembler, 230
arquitetura, 37-40
auxiliar, 39
chaveada em bancos, principal e auxiliar, 214-18
conceitos básicos, 211
descompressão de dados, 187-191
endereços, 3
Merlin, 230
necessária aos assemblers, 229-232
não chaveável, 217
ORCA/M, 230
posições na, 3
principal, 39
Merlin Pro, IX, 59, 72-78, 87, 91, 137, 169
comandos, 73-76
módulos, 72-73

M, indicador, 50
Microcomputadores, arquitetura, 3-6
Microprocessador 6502B/65C02, 13
Microprocessadores, 33-52
Mnemônicos, 2, 11, 106
Modo nível de comando, 55
Monitor, Apple, 87-91
Monitor de linguagem de máquina, 86
Monitor descompilação, 89
Mouse, 17, 247-268
Mouse, modos de operação, 263
MPU, 3
Multiplicação, 198-202
Multiprecisão, divisão binária, 202-204

Negativo, indicador de, 49
Nibble, 20
NOP, 117
Números BCD, 46-47
Números binários, 10, 19-31
manipulação de bits individuais, 178-192
Números com sinal, 204-207
Números em 16 bits, 196-197
Números em ponto flutuante, 208-209
Números hexadecimais, 13, 19-31

Op Code, 65-66
Op Codes 65C02, Tabela de, 323
Operando, 66
ORA, 118
ORA, operador, 185
ORCA/M, IX, 54, 79-84, 85, 91, 137, 169
ORG, 35
Origin directive (ORG), 35
Overflow, indicador de, 48, 207

Página, 212
Página Zero,
endereçamento, 139
endereçamento X, 145
endereçamento Y, 145
endereçamento indireto, 149
Páginas zero, 216-217
Parâmetros opcionais, 69
PHA, 118
PHP, 118
PHX, 118
PHY, 118
P, indicadores, 50-51
Pilha, 15, 149-155, 216-217
Pilha, ponteiro da, 43, 150-151
Pixels, 237
PLA, 119
PLP, 119
PLX, 119
PLY, 119
PREAD, 249
Prefixos para as bases numéricas, 19
PRINT, 58
ProDOS, VIII–X, 137, 169
ProDOS Assembler Tools, 85
ProDOS, Comandos, 68
ProDOS, mapa de memória, 228-229
ProDOS, programação sob assembly, 227-228
Programa em linguagem assembly, escrevendo um, 53-83
Programa "startup", 93-94
Programas de apresentação na tela, 273-280

RAM, 4-5, 33, 295
RAM auxiliar, 213, 215
RAM principal, 213, 215
Registro de banco de dados, 152
Registro de bancos do programa, 52
Registro de status, 44
Registro de status do processador, 44-49
Registros, 52
Registros como contadores, 159
Registros internos, 14
Relativo, endereçamento, 141
Relativos, números de linhas, 56-58, 75
Retorno de carro, 170
Rodando um programa em assembly, 85-104
ROL, 119-183 (rotação)
ROM, 4-5, 33
ROR, 120, 183 (rotação)
RTI, 120
Rotinas automodificáveis, 307
RTS, 120

SAVE, 60
SBC, 121
SEC, 121

SED, 121
SEI, 121
Simulador de Bytes, 124
Sistema operacional, 153
Sistemas numéricos, 19-31
S, Registro, 52
Soft switch, 210
Soft switches, 39, 216
STA, 122
STX, 122
STY, 122
STZ, 122
Sub-rotina, 13, 92-93
Sub-rotinas, 161
Subtração em 16 bits, 197
SWAP, 59

Tabela de Símbolos, 91-93
TAX, 122
Tela, 270-293
  mapeada em bits, 239
Texto, buffer de, 171
Texto e gráfico, modos, 233
Texto em 80 colunas, 176, 236-237
Texto, modos de, 234
TXS, 124
TYA, 124

Unidade Aritmética Lógica (ALU), 40
Unidade Central de Processamento (CPU), 3, 33-52
USR (X), 97-104

Via de Dados, 37-39
Via de Endereços, 37, 38, 39

X, Indicador, 52
X, registro, 42-43

Y, registro, 43

Zero, indicador de, 45

Composição e Arte-Final:
JAG Composição Editorial e Artes Gráficas Ltda.
Praça F. Roosevelt, 208 - 8º andar
Tel. (011) 255-5694 - São Paulo

Impresso na Grol
03043 Rua Martim Burchard, 246
Brás - São Paulo - SP
Fone: (011) 270-4388 (PABX)
com filmes fornecidos pelo Editor

OUTROS LIVROS NA ÁREA

Carvalho	– Assembler para o TK 90X, TK 95
Carvalho	– Assembler para o MSX
Cereda/Maldonado	– Introdução ao Fortran 77 para Microprocessadores
Ciarcia	– Construa seu próprio Microcomputador Z80
Cramer	– 68000 Microprocessador
Malvino	– Microcomputadores e Microprocessadores
Malvino	– Eletrônica Digital
Mitchell	– 32 Bits – Microprocessador
Morgan	– 8086/8088 – Manual do Microprocessador de 16 Bits
Mottola	– Linguagem de Programação Assembly para Apple II – 6502
Osborne	– Introdução aos Microcomputadores – vols. 0 e 1
Shimizu	– Programação Assembler – Z80, 6502, 8080, 8085
Shimizu	– Programação Assembler para Microprocessadores 68000, 68010, 68020
Taub	– Circuitos Digitais e Microprocessadores
Taub	– Eletrônica Digital
Tokheim	– Introdução aos Microprocessadores
Tokheim	– Princípios Digitais
Tokheim	– Circuitos Eletrônicos e de Microcomputadores

0-07-450312-X